

VSIPL++ Specification¹

1.02

CodeSourcery, Inc.

3rd June 2010

¹This specification developed under subcontract 601-02-S-0109 under U.S. Government contract F30602-00-D-0221.

© 2005 by Georgia Tech Research Corporation. All rights reserved

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, distribute, and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies:

THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OR MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

The U.S. Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.

Published 2010

Printed in the United States of America

11 10 09 08 07 06 05 04

5 4 3 2 1

Foreword

The VSIPL++ Library provides C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors. VSIPL++ contains

- containers such as vectors, matrixes, and tensors,
- mathematical operations such as addition and matrix multiplication on these containers,
- complex numbers and random numbers,
- linear algebra solvers such as LU decomposition, and
- signal processing classes and functions including fast Fourier transforms, convolutions, correlations, FIR filters, and IIR filters.

VSIPL++ extends the Vector, Signal, and Image Processing Library (VSIPL) standard, improving application performance, productivity, and portability. This C++ library supports improved performance through support for execution using multiple processors and through improved code optimization. The ability to write expressions using mathematical notation decreases program size, improving productivity. Because VSIPL++ has an open specification and is based on standard C++, VSIPL++ programs can be easily ported between architectures.

The specification and implementation of VSIPL++ has been split into two overlapping development tracks: uniprocessor execution and multi-processor execution. Uniprocessor execution is specified in this document, while distributed execution is specified in a separate document. After specification development finishes, these two documents will be merged together into a single specification.

This main specification document mainly contains declarations, definitions, and requirements for uniprocessor execution. Because VSIPL++ supports both uniprocessor and distributed execution using the same programs, some aspects of distributed execution occur in this document. Some of these items and terms specified in the distributed specification that occasionally appear in this document

include processor, subblock, distribution, and maps. For uniprocessor execution, these distributed aspects can be safely ignored since default arguments are provided by the library.

The VSIPL++ parallel specification is significantly shorter than the uniprocessor specification because the library has been carefully designed to support running the same programs in either single or multiple processor modes. Version 1.0 of the parallel specification (document titled “VSIPL++ Specification – Parallel Specification”, dated 7 April 2006) contains means to specify how a container’s contents are distributed among available processors. Mathematical operations and other operations need not be specified because the toolkit automatically ensures data is moved to where it is used.

Please visit the High Performance Embedded Computing Software Initiative webpage <http://www.hpec-si.org> for more information on VSIPL++ and for reference implementations.

Contents

1	[intro] Introduction	1
1.1	[intro.scope] Scope	1
1.2	[intro.refs] Normative references	1
1.3	[intro.compliance] Compliance	2
1.4	[intro.ack] Acknowledgements	2
2	[conventions] Conventions	3
2.1	[conventions.defs] Definitions	3
2.2	[conventions.struct] Structure	4
2.2.1	[conventions.struct.summary] Summary	4
2.2.2	[conventions.struct.requirements] Requirements	5
2.2.3	[conventions.struct.spec] Specifications	5
2.3	[conventions.requirements] Library-wide requirements	6
3	[support] Support	9
3.1	[support.macros] Macros	10
3.2	[support.types] Types	11
3.2.1	[support.types.domain] Domain types	12
3.2.2	[support.types.dimorder] Dimension-ordering types	13
3.2.3	[support.types.function] Function and class types	14
3.3	[support.exceptions] Exceptions	14
3.3.1	[support.exceptions.comput] Class <code>computation_error</code>	14
3.4	[support.constants] Constants	15
4	[initfin] Initialization and finalization	17
5	[domains] Domains and domain operations	19
5.1	[domains.definitions] Definitions	20
5.1.1	[domains.definitions.subdomain] Subdomain	20
5.1.2	[domains.definitions.overlap] Overlapping domains	20
5.1.3	[domains.definitions.conformant] Conformant domains	20
5.2	[domains.domain] <code>Domain</code>	21
5.2.1	[domains.domain.template] Template parameter	21

5.2.2	[domains.domain.ordering] Lack of ordering	21
5.2.3	[domains.domain.constructors] Constructors, copy, assignment, and destructor	22
5.2.4	[domains.domain.comparison] Comparison operators	22
5.2.5	[domains.domain.accessors] Accessors	23
5.2.6	[domains.domain.equality] Equality functions	23
5.3	[domains.domainone] <code>Domain<1></code>	24
5.3.1	[domains.domainone.position] Index position	25
5.3.2	[domains.domainone.constructors] Constructors	25
5.3.3	[domains.domainone.comparison] Comparison function	26
5.3.4	[domains.domainone.accessors] Accessors	27
5.4	[domains.arithmetic] Arithmetic operations on <code>Domains</code>	27
5.4.1	[domains.arithmetic.shift] Shifting the initial index	28
5.4.2	[domains.arithmetic.scale] Scaling the stride	29
5.5	[domains.index] <code>Index</code>	30
5.5.1	[domains.index.correspond] Correspondence	31
5.5.2	[domains.index.template] Template parameter	31
5.5.3	[domains.index.constructors] Constructors, copy, assignment, and destructor	31
5.5.4	[domains.index.accessors] Accessors	32
5.5.5	[domains.index.equality] Equality functions	32
6	[view] Blocks and views	35
6.1	[view.block] Block requirements	35
6.1.1	[view.block.alloc] Allocatable Block	38
6.2	[view.view] View requirements	39
6.2.1	[view.view.assign] View assignments	41
6.3	[view.dense] Dense block	42
6.3.1	[view.dense.uservocab] User-specified storage vocabulary	44
6.3.2	[view.dense.template] Template parameters	45
6.3.3	[view.dense.constructors] Constructors, copy, assignment, and destructor	46
6.3.4	[view.dense.userdata] User-specified storage	50
6.3.5	[view.dense.valaccess] Value Accessors	55
6.3.6	[view.dense.accessors] Accessors	56
6.4	[view.vector] <code>Vector</code>	57
6.4.1	[view.vector.template] Template parameters	62
6.4.2	[view.vector.subview_types] Subview types	63
6.4.3	[view.vector.constructors] Constructors, copy, assignment, and destructor	63

6.4.4	[view.vector.assign] Assignment operators	66
6.4.5	[view.vector.valaccess] Value accessors	71
6.4.6	[view.vector.subviews] Subviews	72
6.4.7	[view.vector.accessors] Accessors	74
6.4.8	[view.vector.convert] Vector type conversion	74
6.5	[view.matrix] <i>Matrix</i>	74
6.5.1	[view.matrix.template] Template parameters	81
6.5.2	[view.matrix.subview_types] Subview Types	81
6.5.3	[view.matrix.constructors] Constructors, copy, assignment, and destructor	83
6.5.4	[view.matrix.assign] Assignment operators	85
6.5.5	[view.matrix.valaccess] Value accessors	91
6.5.6	[view.matrix.subviews] Subviews	92
6.5.7	[view.matrix.accessors] Accessors	96
6.5.8	[view.matrix.convert] Matrix type conversion	97
6.6	[view.tensor] <i>Tensor</i>	97
6.6.1	[view.tensor.template] Template parameters	107
6.6.2	[view.tensor.subview_types] Subview Types	107
6.6.3	[view.tensor.constructors] Constructors, copy, assignment, and destructor	109
6.6.4	[view.tensor.assign] Assignment operators	112
6.6.5	[view.tensor.transpose] Transpositions	117
6.6.6	[view.tensor.valaccess] Value accessors	118
6.6.7	[view.tensor.subviews] Subviews	118
6.6.8	[view.tensor.accessors] Accessors	129
6.6.9	[view.tensor.convert] Tensor type conversion	130
7	[complex] Complex numbers	131
7.1	[complex.convert] Conversions	132
8	[math] Mathematical functions and operations	135
8.1	[math.enum] Enumerations	135
8.2	[math.definitions] Definitions	136
8.3	[math.fns] Integral, real, complex, and boolean functions	136
8.3.1	[math.fns.promotions] Type Promotions	158
8.3.2	[math.fns.errors] Domain and range errors	158
8.3.3	[math.fns.scalar] Scalar functions	158
8.3.4	[math.fns.elements] Scalar functions used in element-wise ex- tensions	169

8.3.5	[math.fns.elementwise] Scalar functions and their element-wise extensions	181
8.3.6	[math.fns.scalarview] Element-wise functions with scalar and view operands	184
8.3.7	[math.fns.userelt] Element-wise extension of user-specified functions	186
8.3.8	[math.fns.reductions] Reduction functions	196
8.3.9	[math.fns.reductidx] Reduction functions also returning indices	200
8.3.10	[math.fns.operators] Operators	203
8.4	[math.matvec] Matrix and Vector Operations	205
8.4.1	[math.matvec.dot] Dot products	208
8.4.2	[math.matvec.transpose] Matrix transpositions	209
8.4.3	[math.matvec.kron] Kronecker tensor product	210
8.4.4	[math.matvec.outer] Outer product	211
8.4.5	[math.matvec.product] Matrix products	212
8.4.6	[math.matvec.gem] Generalized Matrix operations	217
8.4.7	[math.matvec.vmmul] Vector-Matrix products	219
8.4.8	[math.matvec.misc] Miscellaneous functions	220
8.5	[math.solvers] Linear system solvers	221
8.5.1	[math.solvers.covsol] Covariance linear system solver	223
8.5.2	[math.solvers.llsqsol] Linear least squares solver	224
8.5.3	[math.solvers.toeplitz] Toeplitz linear system solver	226
8.5.4	[math.solvers.lu] LU linear system solver	227
8.5.5	[math.solvers.cholesky] Cholesky decomposition linear system solver	230
8.5.6	[math.solvers.qr] QR decomposition linear system solver	233
8.5.7	[math.solvers.svd] Singular-value decomposition	241
9	[selgen] Selection, generation, and manipulation functions	255
9.1	[selgen.selection] Selection functions	257
9.1.1	[selgen.selection.first] Find leftmost true pair	257
9.1.2	[selgen.selection.indexbool] Find indices of non-false values	258
9.1.3	[selgen.selection.gather] Gathering specified values	258
9.1.4	[selgen.selection.scatter] Scattering specified values	259
9.2	[selgen.generate] Generation functions	260
9.2.1	[selgen.generate.ramp] Filling with a linear function	260
9.3	[selgen.clip] Clipping and inverse clipping	261
9.4	[selgen.manipulation] Manipulation functions	262

10	[random] Random number generation	265
10.1	[random.rand] <code>Rand</code>	265
10.1.1	[random.rand.view_types] View types	266
10.1.2	[random.rand.constructors] Constructors, copy, assignment, and destructor	266
10.1.3	[random.rand.generate] Number generators	267
11	[signal] Signal processing	269
11.1	[signal.hint] Implementation hints	271
11.2	[signal.fft] Single fast Fourier transformations	271
11.2.1	[signal.fft.constants] Constants	274
11.2.2	[signal.fft.template] Template parameters	274
11.2.3	[signal.fft.constructors] Constructors, copy, assignment, and destructor	275
11.2.4	[signal.fft.accessors] Accessors	276
11.2.5	[signal.fft.operators] Operators	276
11.3	[signal.fftm] Multiple fast Fourier transformations	278
11.3.1	[signal.fftm.constants] Constants	280
11.3.2	[signal.fftm.template] Template parameters	281
11.3.3	[signal.fftm.constructors] Constructors, copy, assignment, and destructor	282
11.3.4	[signal.fftm.accessors] Accessors	282
11.3.5	[signal.fftm.operators] Operators	283
11.4	[signal.convolver] Convolutions	285
11.4.1	[signal.convolver.template] Template parameters	286
11.4.2	[signal.convolver.enum] Enumerations	287
11.4.3	[signal.convolver.constructors] Constructors, copy, assignment, and destructor	287
11.4.4	[signal.convolver.accessors] Accessors	287
11.4.5	[signal.convolver.operators] Convolution operators	289
11.5	[signal.correl] Correlations	289
11.5.1	[signal.correl.template] Template parameters	290
11.5.2	[signal.correl.constructors] Constructors, copy, assignment, and destructor	291
11.5.3	[signal.correl.accessors] Accessors	291
11.5.4	[signal.correl.operators] Correlation operators	292
11.6	[signal.windows] Window creation functions	293
11.7	[signal.fir] FIR filters	294
11.7.1	[signal.fir.enum] Enumeration	295
11.7.2	[signal.fir.template] Template parameters	295

CONTENTS

11.7.3	[signal.fir.constructors] Constructors, copy, assignment, and destructor	296
11.7.4	[signal.fir.accessors] Accessors	297
11.7.5	[signal.fir.operators] Filtering and state reset operators	298
11.8	[signal.iir] IIR filters	299
11.8.1	[signal.iir.template] Template parameters	300
11.8.2	[signal.iir.constructors] Constructors, copy, assignment, and destructor	300
11.8.3	[signal.iir.accessors] Accessors	301
11.8.4	[signal.iir.operators] Filtering and state reset operators	302
11.9	[signal.histo] Histograms	302
11.9.1	[signal.histo.constructors] Constructor and destructor	303
11.9.2	[signal.histo.operators] Histogram operators	303
11.10	[signal.freqswap] Frequency swap functions	304
12	[changes] Specification Changes	305

List of Tables

6.1	Block requirements	36
6.2	Modifiable block requirements	38
6.3	View requirements	40
6.4	Modifiable view requirements	40
8.1	Integral, real, complex, and boolean functions with element-wise extensions	182
8.2	Functions with elementwise extensions on views and scalars	186
8.3	Overloaded operators	204
8.4	Overloaded assignment operators	205
8.5	Overloaded logical operators differing according to parameter types	205

1 Introduction

[intro]

1.1 Scope

[intro.scope]

- 1 This document specifies requirements for implementations of the VSIPL++ Library.
- 2 The VSIPL++ Library is a C++ programming interface for performing linear algebra and signal processing computations.
- 3 For classes and class templates, this document specifies partial definitions. Private members are not specified, but each implementation shall supply them to complete the definitions according to the descriptions given herein.
- 4 For functions, function templates, objects, and values, this document specifies complete or partial declarations. Implementations shall supply definitions consistent with the descriptions given herein.

1.2 Normative references

[intro.refs]

- 1 The following standard contains provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and users of this standard are encouraged to investigate the possibility of applying the most recent editions of the standard indicated below.
 - ISO/IEC 14882, Programming languages — C++
- 2 The VSIPL++ Specification — Parallel Specifications is incorporated via reference.
- 3 References to “VSIPL” or the “VSIPL specification” refer to the VSIPL 1.1 API by David A. Schwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manley,

2002 June 11, as approved by the VSIPL Forum, copyright © 1999–2002 Georgia Tech Research Corporation.

1.3 Compliance

[intro.compliance]

- 1 All implementations must define the macro `VSIP_HAS_EXCEPTIONS` in `<vsip/support.hpp>` to a constant-expression such that in:

```
#if VSIP_HAS_EXCEPTIONS
```

`VSIP_HAS_EXCEPTIONS` evaluates to a non-zero value if and only if the implementation throws the exceptions indicated by “Throws” clauses in this specification.

If `VSIP_HAS_EXCEPTIONS` evaluates to zero, then, in all situations where this specification would otherwise require that an exception be thrown, the behavior is undefined. Implementations that do not support exceptions must provide an implementation-defined way to report memory allocation errors to the user.

1.4 Acknowledgements

[intro.ack]

- 1 The VSIPL++ Library as described in this standard is based on the VSIPL 1.1 API by David A. Schwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manley, 2002 June 11, as approved by the VSIPL Forum, copyright © 1999–2002 Georgia Tech Research Corporation.

This clause describes various conventions that apply throughout the remainder of this document.

2.1 Definitions

[conventions.defs]

- 1 Many terms used in this specification are defined in other specifications, and have the same meaning here as in these references.
- 2 All terms defined in ISO/IEC 14882:1998 Programming Languages — C++ have the same meaning in this specification unless otherwise indicated. In particular, these words are defined.

implementation-defined: This phrase means that specifics depend on a particular implementation and that each implementation should document.

undefined: This word means that behavior must meet no specific requirements. Undefined behavior may also be expected when this standard omits a description of any explicitly defined behavior.

unspecified: This word means that specifics depend on a particular implementation.

- 3 All functions, macros, types defined in VSIPL 1.1 API have the same meaning in this specification unless otherwise indicated.
- 4 All terms defined in IETF RFC 2119 have the same meaning in this specification unless otherwise indicated. In particular, these words are defined.

must: This word, or the terms “required” or “shall,” mean that the definition is an absolute requirement of the specification.

must not: This phrase, or the phrase “shall not,” means that the definition is an absolute prohibition of the specification.

should: This word, or the adjective “recommended,” means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

should not: This phrase, or the phrase “not recommended” means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

may: This word, or the adjective “optional,” means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item.

- 5 A *complex type* is a specialization of `complex`. Objects of such types can store complex numbers. A complex type `complex<T>`'s *underlying type* is the numeric type `T`.

2.2 *Structure*

[conventions.struct]

- 1 Each clause contains the following elements, as applicable:¹

- Summary
- Requirements
- Detailed Specifications

2.2.1 *Summary*

[conventions.struct.summary]

- 1 The Summary provides a synopsis of the clause, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.
- 2 Paragraphs labeled “Note(s):”, “Example(s):”, or “Rationale(s):” are informative, other paragraphs are normative.
- 3 The summary and the detailed specifications are presented in the order:

¹To save space, items that do not apply to a clause are omitted. For example, if a clause does not specify any requirements, there will be no “Requirements” subclause.

- Macros
- Values
- Types
- Classes
- Functions
- Objects

2.2.2 *Requirements*

[conventions.struct.requirements]

- 1 VSIP++ can be extended by a C++ program. Each clause, as applicable, describes the requirements that such extensions must meet. Such extensions are generally one of the following:
 - Template arguments
 - Derived classes
 - Containers and/or algorithms that meet an interface convention
- 2 Interface convention requirements are stated as generally as possible. Instead of stating “class X has to define a member function `operator++()`”, the interface requires “for any object `x` of class X, `++x` is defined.” That is, whether the operator is a member is unspecified.
- 3 Requirements are stated in terms of well-defined expressions, which define valid terms of the types that satisfy the requirements. For every set of requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm that uses the requirements is described in terms of the valid expressions for its formal type parameters.
- 4 In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.
- 5 If constraints for a function, function template, object, or value are violated, the function’s, function template’s, object’s, or value’s behavior is undefined.

2.2.3 *Specifications*

[conventions.struct.spec]

- 1 The detailed specifications contain the following elements:

- Name and brief description
- Synopsis (class definition or function prototype, as appropriate)
- Restrictions on template arguments, if any
- Description of class invariants
- Description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):²

- Constructor(s) and destructor
- Copying & assignment functions
- Comparison functions
- Modifier functions
- Operators and other non-member functions

3 Descriptions of function semantics contain the following elements (as appropriate):³

Requires The preconditions for calling the function

Effects The actions performed by the function

Returns A description of the value(s) returned by the function

Postconditions The observable results established by the function

Throws Any exceptions thrown by the function, and the conditions that would cause the exception

2.3 *Library-wide requirements*

[conventions.requirements]

- 1 (ISO14882, [lib.using.headers]/2) is incorporated by reference.
- 2 The `vsip` namespace contains all library functions, classes, types, exceptions, and similar entities. Macros reside outside the namespace.
- 3 The `vsip::impl` namespace, if present, contains functions, classes, types, exceptions, and similar entities useful for the implementation. The contents of this namespace are unspecified.

²To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no “Comparison functions” subclause.

³To save space, items that do not apply to a function are omitted. For example, if a function does not specify any preconditions, there will be no “Requires” paragraph.

- 4 All macros beginning with `VSIP_` are reserved by the implementation.
- 5 For all classes defined by the specification, all class member names beginning with `impl_` are reserved for use by the implementation.
- 6 For the sake of exposition, the specification does not describe copy constructors, assignment operators, and (non-virtual) destructors with the same apparent semantics as those that can be generated by default. It is unspecified whether the implementation provides explicit definitions for such member function signatures or for virtual destructors that can be generated by default.
- 7 Implementation header files shall have internal include-guards to protect against multiple inclusion of a single header. If a file is included multiple times, only the first inclusion will be processed.

- 1 This clause describes basic macros, types, exceptions, and similar entities that are used throughout the library.

Header <vsip/support.hpp> synopsis

```
namespace vsip {
    typedef implementation-defined VSIP_DEFAULT_VALUE_TYPE;
    typedef implementation-defined cscalar_f;
    typedef implementation-defined cscalar_i;
    typedef implementation-defined dimension_type;
    typedef implementation-defined scalar_f;
    typedef implementation-defined scalar_i;

    // domain types
    typedef implementation-defined index_type;
    typedef signed-version-of-index_type index_difference_type;
    typedef signed-version-of-index_type stride_type;
    typedef unsigned-version-of-stride_type stride_scalar_type;
    typedef index_type length_type;
    | enum whole_domain_type { whole_domain };

    // dimension ordering
    template <dimension_type dim0 = unspecified,
        ...,
        dimension_type dimn = unspecified>
        class tuple;
    // exactly VSIP_MAX_DIMENSION template parameters
    typedef tuple<0, unspecified, ...> row1_type;
    typedef tuple<0, 1, unspecified, ...> row2_type;
    // ...
    // exactly VSIP_MAX_DIMENSION type definitions
    typedef tuple<0, unspecified, ...> col1_type;
    typedef tuple<1, 0, unspecified, ...> col2_type;
    // ...
}
```

3.1. [SUPPORT.MACROS] MACROS

```
// exactly VSIP_MAX_DIMENSION type definitions

// map and processor types
typedef implementation-defined processor_type;
typedef implementation-defined subblock_type;
typedef signed-version-of-processor_type processor_difference_type;
typedef signed-version-of-subblock_type subblock_difference_type;
enum distribution_type { block, cyclic };

// function and class types
enum return_mechanism_type { by_value, by_reference };

// exceptions
struct computation_error;

// functions
processor_type num_processors () VSIP_NOTHROW;

// dimension_type synonyms
dimension_type const row = 0;
dimension_type const col = 1;
// Exactly VSIP_MAX_DIMENSION constants:
dimension_type const dim0 = 0;
dimension_type const dim1 = 1;
dimension_type const dim2 = 2;
// ...
}
```

3.1 Macros

[support.macros]

```
1 #define VSIP_MAX_DIMENSION implementation-defined
```

Value: VSIP_MAX_DIMENSION indicates the maximum dimension supported by blocks in the VSIPL++ library. The library implementation must provide blocks, views, and domains for all dimensions less than or equal to VSIP_MAX_DIMENSIONS. However, some entities may support more than VSIP_MAX_DIMENSION dimensions.

VSIP_MAX_DIMENSION shall be an unsigned integral quantity greater than or equal to two. The value shall be suitable for use as a constant-expression in preprocessing directives.

[*Note:* VSIP_MAX_DIMENSION should be a positive integral literal value. It must not expand to the name of a const variable.

C++ does not allow:

```
int const __vsip_max_dimension = 3;
#define VSIP_MAX_DIMENSION __vsip_max_dimension
#if VSIP_MAX_DIMENSION > 2
    // Conditionally compiled code.
#endif
]
```

2 #define VSIP_HAS_EXCEPTIONS *implementation-defined*

Value: Evaluates to a non-zero value if and only if the implementation throws the exceptions indicated by “Throws” requirements. [*Note:* [intro.compliance] specifies the interaction of VSIP_HAS_EXCEPTIONS with exceptions.]

3 #define VSIP_NOTHROW *implementation-defined*

Value: VSIP_NOTHROW behaves as if it has been defined either as #define VSIP_NOTHROW or as #define VSIP_NOTHROW throw() .
[*Note:* C++ implementations that do not provide adequate support for throw specifications should provide an empty definition.]

4 #define VSIP_THROW(X) *implementation-defined*

Value: VSIP_THROW behaves as if it has been defined either as #define VSIP_THROW(X) or as #define VSIP_THROW(X) throw X.
[*Note:* C++ implementations that do not provide adequate support for throw specifications should provide an empty definition.]

3.2 Types

[support.types]

1 typedef *implementation-defined* VSIP_DEFAULT_VALUE_TYPE;

Value: VSIP_DEFAULT_VALUE_TYPE indicates the default type of values when creating Vectors, Matrixes, and Tensors.

2 typedef *implementation-defined* cscalar_f;

Value: The implementation shall define this type to be complex<float>, complex<double>, or complex<long double> such that the choice of underlying type is the same as scalar_f .

3.2.1. [SUPPORT.TYPES.DOMAIN] DOMAIN TYPES

3 typedef *implementation-defined* cscalar_i;

Value: The implementation shall define this type to be `complex<short int>`, `complex<int>`, `complex<long int>`, `complex<unsigned short int>`, `complex<unsigned int>`, `complex<unsigned long int>`, `complex<long long int>`, or `complex<unsigned long long int>` such that the choice of underlying type is the same as `scalar_i`.

4 typedef *implementation-defined* dimension_type;

Value: The `dimension_type` type shall be an unsigned integral type.
The type shall be chosen such that `VSIP_MAX_DIMENSION` can be implicitly converted to `dimension_type`.

5 typedef *implementation-defined* scalar_f;

Value: The implementation shall define this type to be `float`, `double`, or `long double`.

6 typedef *implementation-defined* scalar_i;

Value: The implementation shall define this type to be `short int`, `int`, `long int`, `unsigned short int`, `unsigned int`, `unsigned long int`, `long long int`, or `unsigned long long int`.

3.2.1 Domain types

[support.types.domain]

1 typedef *implementation-defined* index_type;

Value: The `index_type` type is an unsigned integral type indicating one coordinate of an `Index<D>` in a `Domain<D>` for a fixed `dimension_type D`. This type must support values large enough to represent the largest size of a view.

Note: `index_type` can be `size_t`.

2 typedef *signed-version-of-index_type*
index_difference_type;

Value: The `index_difference_type` type is a signed integer representing the difference of two `index_types`, e.g., incrementing or decrementing an `index_type`.

Note: Domain arithmetic uses this type.

3 `typedef signed-version-of-index_type stride_type;`

Value: The `stride_type` type is a signed version of `index_type` indicating the stride between successive indices along a dimension.

4 `typedef unsigned-version-of-stride_type stride_scalar_type;`

Value: The `stride_scalar_type` type is an unsigned multiple of a stride.

Note: Domain arithmetic uses this type.

5 `typedef index_type length_type;`

Value: A number of indices.

6 The `whole_domain_type` enumerated value `whole_domain` is used to indicate that an entire domain should be returned by certain Tensor subview functions.

3.2.2 *Dimension-ordering types*

[support.types.dimorder]

1 `tuple` requires exactly `VSIP_MAX_DIMENSION` template parameters.

2 Exactly `VSIP_MAX_DIMENSION` type definitions `row1_type`, `row2_type`, ... are required. For positive `D` at most `VSIP_MAX_DIMENSION`, `rowD_type`'s first `D` template values are 0, 1, ..., `D-1`. The remaining `VSIP_MAX_DIMENSION-D` entries are unspecified.

[*Note:* These represent row-major ordering for a one-dimensional block, two-dimensional block, etc.]

3 Exactly `VSIP_MAX_DIMENSION` type definitions `col1_type`, `col2_type`, ... are required. For positive `D` at most `VSIP_MAX_DIMENSION`, `colD_type`'s first `D` template values are `D-1`, ..., 1, 0. The remaining `VSIP_MAX_DIMENSION-D` entries are unspecified.

[*Note:* These represent column-major ordering for a one-dimensional block, two-dimensional block, etc.]

3.2.3 *Function and class types*

[support.types.function]

- 1 [*Note:* The `return_mechanism_type` enumerated value `by_value` indicates a function returns a computed value. `by_reference` indicates a function requires a parameter where the computed value is saved.]

3.3 *Exceptions*

[support.exceptions]

- 1 This section specifies some of the exceptions that VSIPL++ functions and objects can throw. [*Note:* Some errors are indicated by throwing exceptions.]
- 2 Memory allocation errors will be indicated by `std::bad_alloc` exceptions. [*Note:* VSIPL 1.1 API [p. 11] defines errors, which are fatal in development mode and which produce undefined results in production mode. This specification does not define a development mode or a production mode. Instead, it defines exceptions that are thrown by supporting implementations. Memory allocation failures, which are indicated by the function return value in VSIPL, are indicated by throwing the `std::bad_alloc` exception.

As noted in [intro.compliance], implementations that do not support exceptions must provide an implementation-defined way to report memory allocation errors to the user.]

3.3.1 *Class computation_error*

[support.exceptions.comput]

```
namespace vsip {  
    class computation_error : public std::runtime_error {  
    public:  
        explicit computation_error(std::string const&);  
    };  
}
```

- 1 The class `computation_error` defines the type of objects thrown as exceptions to report errors presumably caused by algorithmic computations and detectable only when the program executes.

```
computation_error(std::string const& what_arg);
```

Effects: Constructs an object of class `computation_error`.

Postcondition: `strcmp(this->what(), what_arg.c_str()) == 0`.

3.4 Constants

[support.constants]

- 1 [*Note:* `row` and `col` correspond to VSIPL constants `VSIP_ROW` and `VSIP_COL`.]
- 2 An implementation must define `VSIP_MAX_DIMENSIONS` `dimension_type` constants `dim0`, `dim1`, `dim2`,

-
- 1 This clause describes the `vsipl` class. At least one `vsipl` object must exist while using the library.

Header <vsip/initfin.hpp> synopsis

```
namespace vsip {
  class vsipl {
  public:
    vsipl();
    vsipl(int& argc, char**& argv);
    ~vsipl();
  private:
    vsipl(vsipl const&);
    vsipl& operator=(vsipl const&);
  };
}
```

- 2 [*Note:* This header file is usually included in other header files so direct inclusion is rarely necessary.]

```
vsipl();
```

Effects: If there are no other extant `vsipl` objects, initializes the library before its use. It behaves as if it were implemented as `if (counter++ != 0) return; else /* perform initialization */ ...`

Note: Failure to correctly initialize the library leads to undefined behavior. This behavior can include throwing an exception.

```
vsipl(int& argc, char**& argv);
```

Requires: `argc` is the nonnegative number of program arguments in `argv`.
`argv[argc] == 0`.

4. [INITFIN] INITIALIZATION AND FINALIZATION

Effects: If there are no other extant `vsipl` objects, initializes the library before its use. It behaves as if it were implemented as `if (counter++ != 0) return; else /* perform initialization */`

Program arguments used to initialize the library will be removed so the requirements on `argc` and `argv` are still valid.

Note: Failure to correctly initialize the library leads to undefined behavior. This behavior can include throwing an exception.

```
~vsipl();
```

Effects: If there is only one extant `vsipl` object, cleans up library data structures. Otherwise, the destructor has no effect. It behaves as if it were implemented as `if (--counter != 0) return; else /* perform destruction */`

Postconditions: If the destructor had an effect, the VSIPL++ library cannot be used.

Note: Failure of the library to correctly deallocate its resources leads to undefined behavior. This behavior can include throwing an exception.

3 *[Example:* Many VSIPL++ programs will be similar to:

```
int main() {
    vsip::vsipl v;

    // Use VSIPL++ library.
}
```

]

-
- 1 In this clause, unless otherwise specified, D represents a fixed `dimension_type` greater than 0 and at most `VSIP_MAX_DIMENSION`.
 - 2 An `Index<D>` represents an element of \mathbf{N}^D , where \mathbf{N} is the set of nonnegative integers and D is a positive integral dimension. A `Domain<D>` represents a subset of \mathbf{N}^D .
 - 3 Below, the `Domain` class template and its one-dimensional specialization are presented. Subsequent subclauses present arithmetic operations on `Domains`, which facilitate creating subviews of existing views, comparison operators, and the `Index` class template.

Header <vsip/domain.hpp> synopsis

```
namespace vsip {
    template <dimension_type D> class Domain;

    template <dimension_type D>
        bool operator==(Domain<D> const&, Domain<D> const&)
            VSIP_NOTHROW;
    template <dimension_type D>
        bool operator!=(Domain<D> const&, Domain<D> const&)
            VSIP_NOTHROW;

    template <dimension_type D>
        Domain<D> operator+(Domain<D> const&, index_difference_type)
            VSIP_NOTHROW;
    template <dimension_type D>
        Domain<D> operator+(index_difference_type, Domain<D> const&)
            VSIP_NOTHROW;
    template <dimension_type D>
        Domain<D> operator*(Domain<D> const&, stride_scalar_type)
            VSIP_NOTHROW;
    template <dimension_type D>
```

5.1. [DOMAINS.DEFINITIONS] DEFINITIONS

```
    Domain<D> operator*(stride_scalar_type, Domain<D> const&)
    VSIP_NOTHROW;

template <dimension_type D> class Index;

template <dimension_type D>
    bool operator==(Index<D> const&, Index<D> const&)
    VSIP_NOTHROW;
template <dimension_type D>
    bool operator!=(Index<D> const&, Index<D> const&)
    VSIP_NOTHROW;
}
```

5.1 Definitions [domains.definitions]

5.1.1 Subdomain [domains.definitions.subdomain]

- 1 A Domain s is a *subdomain* of another Domain d if every Index in s is also an Index in d .

5.1.2 Overlapping domains [domains.definitions.overlap]

- 1 Two Domains *overlap* if there is at least one Index in both Domains.
- 2 Two Domains *exactly overlap* if every Index of one Domain is in the other Domain and vice versa.

5.1.3 Conformant domains [domains.definitions.conformant]

- 1 Domain<D>s d_1 and d_2 are *element conformant* if $d_1.element_conformant(d_2) == true$. [Note: Intuitively, d_1 and d_2 are element conformant if, for every dimension, they have the same number of indices.]
- 2 Domain<2>s d_1 and d_2 are *product conformant* if $d_1.product_conformant(d_2) == true$. [Note: Product-conformant matrices can be multiplied by each other. d_1 and d_2 are conformant if the second dimension of d_1 has the same length as the first dimension of d_2 . Product-conformance is not commutative.]

5.2 *Domain***[domains.domain]**

- 1 A `Domain` object specifies a nonempty domain, i.e., a set of `Indexes`. A `Domain` is the Cartesian product of one-dimensional `Domain<1>`s.

```

namespace vsip {
  template <dimension_type D>
  class Domain {
  public:
    // compile-time values
    static dimension_type const dim = D;

    // constructors, copy, assignment, and destructor
    Domain() VSIP_NOTHROW;
    Domain(Domain<1> const&, ..., Domain<1> const&)
      VSIP_NOTHROW;
    // exactly D parameters
    Domain(Domain const&) VSIP_NOTHROW;
    Domain& operator=(Domain const&) VSIP_NOTHROW;
    ~Domain() VSIP_NOTHROW;

    // comparison functions
    bool element_conformant(Domain const&) const
      VSIP_NOTHROW;
    bool product_conformant(Domain<2> const&) const
      VSIP_NOTHROW;

    // accessors
    Domain<1> const& operator[](dimension_type) const
      VSIP_NOTHROW;
    length_type size() const VSIP_NOTHROW;
  };
}

```

5.2.1 *Template parameter***[domains.domain.template]**

dimension_type D

Requires: $0 < D \leq \text{VSIP_MAX_DIMENSION}$. A VSIP++ implementation must implement `Domain<D>` for all valid dimensions.

5.2.2 *Lack of ordering***[domains.domain.ordering]**

- 1 `class Domain<D>`, for $D > 1$, does not impose any ordering among its one-dimensional objects.

5.2.3 *Constructors, copy, assignment, and destructor* **[domains.domain-constructors]**

Domain() VSIP_NOTHROW;

Effects: Constructs a D-dimensional domain with one index, i.e., $(0, 0, \dots, 0)$.

Domain(Domain<1> const&, ..., Domain<1> const&) VSIP_NOTHROW;

Notation: The parameter list contains D “const Domain<1>&” parameters.

Requires: All D arguments must be nonempty.

Effects: Constructs a D-dimensional domain that is the Cartesian product of the D given arguments, in the given order. For example, the first argument specifies the permissible set of first coordinates.

Domain(Domain const& d) VSIP_NOTHROW;

Effects: Constructs a D-dimensional domain.

Postconditions: The Domain has `this->operator[] (0) == d[0], ..., this->operator[] (D-1) == d[D-1]`.

Domain&

operator=(Domain const& d) VSIP_NOTHROW;

Effects: Modifies the D-dimensional domain.

Postconditions: The domain has `this->operator[] (0) == d[0], ..., this->operator[] (D-1) == d[D-1]`.

Returns: *this.

5.2.4 *Comparison operators* **[domains.domain.comparison]**

bool

element_conformant(Domain const& d) const VSIP_NOTHROW;

Returns: True if and only if, for all $0 \leq \text{dim} < D$, `(*this)[dim].length() == d[dim].length()`.

bool

product_conformant(Domain<2> const& d) const VSIP_NOTHROW;

Returns: True if and only if $D == 2$ and $(*this)[1].length() == d[0].length()$.

Note: Product conformance between domains is only possible for `Domain<D>` objects where $D==2$; however, this function is defined for `Domain<D>` objects with other values of D to provide a consistent interface among all `Domain` objects.

5.2.5 Accessors

[domains.domain.accessors]

```
Domain<1> const&
operator[](dimension_type dim) const VSIP_NOTHROW;
```

Requires: $dim < D$.

Returns: The one-dimensional domain object for the specified dimension.

```
length_type
size() const VSIP_NOTHROW;
```

Returns: The number of indices. This is the product of the number of indices for each of its dimensions.

5.2.6 Equality functions

[domains.domain.equality]

```
namespace vsip {
    template <dimension_type D>
    bool operator==(Domain<D> const&, Domain<D> const&)
        VSIP_NOTHROW;
    template <dimension_type D>
    bool operator!=(Domain<D> const&, Domain<D> const&)
        VSIP_NOTHROW;
}
```

```
template <dimension_type D>
bool
operator==(
    Domain<D> const& dom0,
    Domain<D> const& dom1)
VSIP_NOTHROW;
```

Returns: true if and only if the Indexes of `dom0` are exactly the same as the Indexes of `dom1`.

Notes: Two Domains can be equal even if `dom0.first() != dom1.first()`. [Example: `Domain<1>(0, 1, 16) == Domain<1>(15, -1, 16)` because they contain exactly the same indices but in a different order.]

```
template <dimension_type D>
bool
operator!=(
    Domain<D> const& dom0,
    Domain<D> const& dom1)
VSIP_NOTHROW;
```

Returns: `!operator==(dom0, dom1)`.

5.3 Domain<1>

[domains.domainone]

- 1 A Domain<1> object specifies one `dimension_type` of a Domain object. This “subscript triplet” specifies the first `index_type` `i`, a `stride_type` `s` between indices, and the number `len` of indices together representing indices `i`, `i+s`, `i+2s`, ..., `i+(len-1)s`. [Note: A subscript triplet is similar to Matlab and Fortran 95 ranges and VSIP view slices. The object maintains the specified subscript triplet even though other triplets may represent the same set of indices.]
- 2 The stride may assume any integral value. Using a positive stride yields a sequence of increasing indices. Using a negative stride yields a sequence of decreasing indices. To produce a sequence of repeated indices, use a stride of zero.

```
namespace vsip {
    template <>
    class Domain<1> {
    public:
        // compile-time values
        static dimension_type const dim = 1;

        // constructors, copy, assignment, and destructor
        Domain(index_type, stride_type, length_type) VSIP_NOTHROW;
        Domain(length_type = 1) VSIP_NOTHROW;
        Domain(Domain const&) VSIP_NOTHROW;
        Domain& operator=(Domain const&) VSIP_NOTHROW;
        ~Domain() VSIP_NOTHROW;

        // comparison function
```

```

bool element_conformant(Domain const&) const
    VSIP_NOTHROW;
bool product_conformant(Domain<2> const&) const
    VSIP_NOTHROW;

// accessors
Domain<1> const& operator[](dimension_type) const
    VSIP_NOTHROW;
index_type first() const VSIP_NOTHROW;
stride_type stride() const VSIP_NOTHROW;
length_type length() const VSIP_NOTHROW;
length_type size() const VSIP_NOTHROW;
};
}

```

5.3.1 Index position

[domains.domainone.position]

- 1 The *position* of an `index_type` `i` within a `Domain<1>` `dom` is $(i - \text{dom.first}()) / \text{dom.stride}()$.

[*Note:* Since the `index_type` is within the `Domain<1>`, its position is integral, at least zero, and less than the Domain's length.

For ordinary C arrays, array indices are numbered $0, 1, \dots, n-1$. When specifying a subscript triplet, the initial value need not be zero and the stride need not be one. Regardless, `index_types` are ordered. Positions represent this ordering. A subscript triplet with first `index_type` `f`, `stride_type` `s` between indices, and the number `len` of indices has `index_types` $f + 0*s, f + 1*s, f + 2*s, \dots, f + (\text{len}-1)*s$ and positions $0, 1, 2, \dots, \text{len}-1$.]

[*Example:* For a `Domain<1>` object specifying a subscript triplet with first `index_type` `f`, `stride_type` `s` between indices, and the number `len` of indices, the position of `index_type` `f` is 0 and (assuming `len >= 3`) of `index_type` `f+2*s`, 2.]

5.3.2 Constructors

[domains.domainone.constructors]

Domain(`index_type` `i`, `stride_type` `s`, `length_type` `len`) VSIP_NOTHROW;

Requires: $\text{len} > 0$. $i + (\text{len} - 1) * s \geq 0$.

Effects: Constructs a one-dimensional domain with indices $i, i + s, i + 2 * s, \dots, i + (\text{len} - 1) * s$.

Postconditions: `this->first()` == `i`, `this->stride()` == `s`, and `this->length()` == `len`.

Notes: Negative strides are supported.

```
Domain(length_type len = 1) VSIP_NOTHROW;
```

Requires: len > 0.

Returns: Domain(0, 1, len).

```
Domain(Domain const& d) VSIP_NOTHROW;
```

Effects: Constructs a one-dimensional domain.

Postconditions: this->first() == d.first(), this->stride() == d.stride(), and this->length() == d.length().

```
Domain&
```

```
operator=(Domain const& d) VSIP_NOTHROW;
```

Effects: Modifies the one-dimensional domain so this->first() == d.first(), this->stride() == d.stride(), and this->length() == d.length().

Returns: *this.

5.3.3 Comparison function

[domains.domainone.comparison]

```
bool
```

```
element_conformant(Domain const& d) const VSIP_NOTHROW;
```

Returns: True if and only if (*this)[0].length() == d[0].length().

```
bool
```

```
product_conformant(Domain<2> const& d)  
const VSIP_NOTHROW;
```

Returns: false.

Note: Product conformance between domains is only possible for Domain<D> objects where D==2; however, this function is defined to provide a consistent interface among all Domain objects.

5.3.4 Accessors

[domains.domainone.accessors]

```
Domain<1> const&
operator[](dimension_type dim) const VSIP_NOTHROW;
```

Requires: `dim < 1`.**Returns:** `*this`.**Note:** This member function provides compatibility with higher dimension Domain types but is otherwise uninteresting.

```
index_type
first() const VSIP_NOTHROW;
```

Returns: The “first” entry of the subscript triplet. Equivalent to `i`, as defined in [domains.domainone]/1. [*Note:* If the stride is nonnegative, this index is the smallest index represented by the triplet. If the stride is negative, this index is the largest index.]

```
stride_type
stride() const VSIP_NOTHROW;
```

Returns: The stride of the subscript triplet. Equivalent to `s`, as defined in [domains.domainone]/1.

```
length_type
length() const VSIP_NOTHROW;
```

Returns: The number of indices in the domain. Equivalent to `len`, as defined in [domains.domainone]/1.

```
length_type
size() const VSIP_NOTHROW;
```

Returns: `this->length()`.

5.4 Arithmetic operations on Domains

[domains.arithmetic]

- 1 To facilitate creation of subviews, Domains support integral arithmetic operations. A Domain’s minimal index can be shifted, and its stride can be multiplied. For example, adding the integer one to a domain increments all index components by one.

```

namespace vsip {
    // shift the initial index
    template <dimension_type D>
    Domain<D> operator+(Domain<D> const&, index_difference_type)
        VSIP_NOTHROW;
    template <dimension_type D>
    Domain<D> operator+(index_difference_type, Domain<D> const&)
        VSIP_NOTHROW;
    template <dimension_type D>
    Domain<D> operator-(Domain<D> const&, index_difference_type)
        VSIP_NOTHROW;

    // modify the stride
    template <dimension_type D>
    Domain<D> operator*(Domain<D> const&, stride_scalar_type)
        VSIP_NOTHROW;
    template <dimension_type D>
    Domain<D> operator*(stride_scalar_type, Domain<D> const&)
        VSIP_NOTHROW;
    template <dimension_type D>
    Domain<D> operator/(Domain<D> const&, stride_scalar_type)
        VSIP_NOTHROW;
}

```

5.4.1 *Shifting the initial index***[domains.arithmetic.shift]**

```

template <dimension_type D>
Domain<D>
operator+(
    Domain<D> const&      dm,
    index_difference_type difference)
VSIP_NOTHROW;

```

Requires: $dm.first() + difference \geq 0$. $dm.first() + difference + dm.stride() * (dm.length() - 1) \geq 0$.

Returns: A domain *dom* equivalent to *dm* except having $dom[d].first() == dm[d].first() + difference$ for all $0 \leq d < D$.

Notes: The resulting domain must have $this->first() \geq 0$.

Examples: `Domain<1> d(2,2,2); (d+1).first()` yields 3.
`Domain<2>(Domain<1>(3,1,4), 0) + 2` contains $\{(5,2), (6,2), (7,2), (8,2)\}$.

```

template <dimension_type D>

```

```

Domain<D>
operator+(
    index_difference_type difference,
    Domain<D> const&          dm)
VSIP_NOTHROW;

```

Requires: $dm.first() + difference \geq 0$.

Returns: `operator+(dm, difference)`.

```

template <dimension_type D>
Domain<D>
operator-(
    Domain<D> const&          dm,
    index_difference_type difference)
VSIP_NOTHROW;

```

Requires: $dm.first() + -difference \geq 0$.

$dm.first() + (-difference) + dm.stride() * (dm.length() - 1) \geq 0$.

Returns: `operator+(dm, -difference)`.

5.4.2 *Scaling the stride*

[domains.arithmetic.scale]

```

template <dimension_type D>
Domain<D>
operator*(
    Domain<D> const&          dm,
    stride_scalar_type str)
VSIP_NOTHROW;

```

Requires: $dm.first() + dm.stride() * str * (dm.length() - 1) \geq 0$.

Returns: A domain `dom` equivalent to `dm` except having `dom[d].stride() == dm[d].stride() * str` for all $0 \leq d < D$.

Examples: `Domain<1> d(2, 3, 2); (d*2).stride()` yields 6.
`Domain<2>(Domain<1>(3, 1, 4), 0) * 2` contains $\{(3, 0), (5, 0), (7, 0), (9, 0)\}$.

```

template <dimension_type D>

```

```
Domain<D>
```

```
operator* (  
    stride_scalar_type  str,  
    Domain<D> const&    dm)  
VSIP_NOTHROW;
```

Requires: $dm.first() + dm.stride() * str * (dm.length() - 1) \geq 0$.

Returns: $dm * str$.

```
template <dimension_type D>  
Domain<D>  
operator/ (  
    Domain<D> const&    dm,  
    stride_scalar_type  str)  
VSIP_NOTHROW;
```

Returns: A domain dom equivalent to dm except having $dom[d].stride() == dm[d].stride() / str$ for all $0 \leq d < D$.

Examples: `Domain<1> d(2,3,2); (d/2).stride()` yields
1. `Domain<2>(Domain<1>(3,4,4), 0) / 2` contains
`{(3,0), (5,0), (7,0), (9,0)}`.

5.5 Index

[domains.index]

- 1 An Index object specifies an element in a Domain, i.e., an ordered set of coordinates — equivalently an ordered set of `index_types`.

```
namespace vsip {  
    template <dimension_type D>  
    class Index {  
    public:  
        // compile-time values:  
        static dimension_type const dim = D;  
  
        // constructors, copy, assignment, and destructor  
        Index() VSIP_NOTHROW;  
        Index(index_type, ..., index_type) VSIP_NOTHROW;  
        // exactly D parameters  
        Index(Index const&) VSIP_NOTHROW;  
        Index& operator=(Index const&) VSIP_NOTHROW;
```

```

    ~Index() VSIP_NOTHROW;

    // accessors:
    index_type operator[](dimension_type) const VSIP_NOTHROW;
};
}

```

5.5.1 Correspondence

[domains.index.correspond]

- 1 $\text{Index}\langle D \rangle (i_0, i_1, \dots, i_{D-1})$ and $\text{Index}\langle D \rangle (j_0, j_1, \dots, j_{D-1})$ correspond if, for all dimensions $d \in [0, D)$, the position of i_d within its dimension- d domain is the same as the position of j_d within its dimension- d domain. [Note: Two element-conformant domains have corresponding Indexes.]

5.5.2 Template parameter

[domains.index.template]

dimension_type D

Requires: $0 < D \leq \text{VSIP_MAX_DIMENSION}$. A VSIP++ implementation must implement $\text{Index}\langle D \rangle$ for all valid dimensions.

5.5.3 Constructors, copy, assignment, and destructor

[domains.index.-constructors]

Index() VSIP_NOTHROW;

Effects: Constructs a D-dimensional Index representing $(0, 0, \dots, 0)$.

Index(index_type, ..., index_type) VSIP_NOTHROW;

Notation: The parameter list contains D index_type parameters.

Effects: Constructs a D-dimensional Index, using parameters in the given order. For example, the first argument specifies the first coordinate.

Note: index_type and $\text{Index}\langle 1 \rangle$ are distinct types.

Index(Index const& i) VSIP_NOTHROW;

Effects: Constructs a D-dimensional Index.

Postconditions: The index has $\text{this}\rightarrow\text{operator}[] (0) == i[0]$, ..., $\text{this}\rightarrow\text{operator}[] (D-1) == i[D-1]$.

```
Index&  
operator=(Index const& i) VSIP_NOTHROW;
```

Effects: Modifies the D-dimensional Index .

Postconditions: The index has `this->operator[] (0) == i[0], ..., this->operator[] (D-1) == i[D-1]` .

Returns: `*this` .

5.5.4 Accessors

[domains.index.accessors]

```
index  
operator [] (dimension_type dim) const VSIP_NOTHROW;
```

Requires: `dim < D` .

Returns: The coordinate for the specified dimension.

5.5.5 Equality functions

[domains.index.equality]

```
namespace vsip {  
    // compare Indexes  
    template <dimension_type D>  
    bool operator==(Index<D> const&, Index<D> const&)  
        VSIP_NOTHROW;  
    template <dimension_type D>  
    bool operator!=(Index<D> const&, Index<D> const&)  
        VSIP_NOTHROW;  
}
```

```
template <dimension_type D>  
bool  
operator==(  
    Index<D> const& idx0,  
    Index<D> const& idx1)  
VSIP_NOTHROW;
```

Returns: `true` if and only if, for all dimensions $d \in [0, D)$, `idx0[d] == idx1[d]` .

```
template <dimension_type D>  
bool  
operator!=(  

```

```
    Index<D> const& idx0,  
    Index<D> const& idx1)  
VSIP_NOTHROW;
```

Returns: !operator==(idx0, idx1).

-
- 1 This clause describes components that VSIPL++ programs may use to store and use data. A block is an interface to a logically contiguous array of data. The `Dense` class is a block. A view is an interface supporting data-parallel operations. `Vector`, `Matrix`, and `Tensor` classes satisfy this interface.
 - 2 A map specifies how a block can be divided into subblocks. [*Note:* For a program executing on a single processor, there should be no need to indicate any particular map since default template arguments and default function arguments should suffice. A VSIPL++ implementation restricted to supporting only a single processor will probably just define empty map classes and will probably not define the view constructors in `[view.vector.vectormap]`, `[view.matrix.matrixmap]`, and `[view.tensor.tensormap]`.]

6.1 Block requirements

[view.block]

- 1 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.]
- 2 A block is said to be *x-dimensional* (where x is a positive integer) if it supports x -dimensional access. An *x,y-dimensional block* may be both x -dimensional and y -dimensional, where x and y are distinct positive integers. [*Note:* For example, a single block may support both 1- and 2-dimensional access. A 1,2-dimensional block can be used as the underlying storage for both vector and matrix views.]
- 3 The type of objects stored in these components must meet the same requirements as types specified for numeric types (ISO14882, `[lib.numeric.requirements]`). [*Note:* (ISO14882, `[lib.numeric.requirements]/2`) states that, if any operation on such a type throws an exception, the effects are undefined.]

- 4 A block may store only one type of values.
- 5 Blocks may be modifiable or non-modifiable. The data of a *modifiable block* can be changed; the data of a *non-modifiable block* cannot be changed.
- 6 Every block shall satisfy the requirements in Table 6.1. In Table 6.1, B denotes an x -dimensional block class containing objects of type T, b denotes a value of type B, X denotes a `dimension_type` instance with value x , i_1, \dots, i_x denote x values of type `index_type`, d denotes a value of type `dimension_type`, and M denotes a map type.

Table 6.1: Block requirements

expression	return type	assertion/note pre/post-condition
<code>B::value_type</code>	T	
<code>B::reference_type</code>	lvalue of T	
<code>B::const_reference_type</code>	const lvalue of T	
<code>B::map_type</code>	M	
<code>b.get(i_1, i_2, ..., i_x)</code>	<code>value_type</code>	
<code>b.size()</code>	<code>length_type</code>	post: product of size(d) over all $0 \leq d < x$
<code>b.size(X, d)</code>	<code>length_type</code>	pre: $0 \leq d < x$
<code>b.increment_count()</code>	none	
<code>b.decrement_count()</code>	none	
<code>b.map()</code>	reference to a M	post: returns a map object

- 7 [Note: The above requirements imply that an x,y -dimensional block will have two `get` functions; one with x parameters and one with y parameters.]

- 8 For an x,y -dimensional block b, let Y represent a `dimension_type` instance with value y and j_0, \dots, j_{y-1} denote y values of type `index_type`. Let $d_0^x, d_1^x, \dots, d_{x-1}^x$ represent the block's x -dimensional dimension-ordering. and $d_0^y, d_1^y, \dots, d_{y-1}^y$ represent the block's y -dimensional dimension-ordering. If and only if offsets:

$$(i_{d_0^x} * b.size(X, d_1^x) * \dots * b.size(X, d_{x-1}^x)) +$$

$$(i_{d_1^x} * b.size(X, d_2^x) * \dots * b.size(X, d_{x-1}^x)) +$$

$$\dots + i_{d_{x-1}^x}$$

and

$$(j_{d_0^y} * b.size(Y, d_1^y) * \dots * b.size(Y, d_{y-1}^y)) +$$

	0	1	2	3
0	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
1	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>

Figure 6.1: A two-dimensional view of a 1,2-dimensional block. Each rectangle contains a value. Row and column indices are labelled on the left and top, respectively.

	0	1	2	3	4	5	6	7
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>

Figure 6.2: A one-dimensional view of a 1,2-dimensional block. Each rectangle contains a value. Value indices are labelled along the top.

$$(j_{d_1^y} * \text{b.size}(Y, d_2^y) * \dots * \text{b.size}(Y, d_{y-1}^y)) + \dots + j_{d_{y-1}^y},$$

are equal, then accessors `b.get(i0, i1, ..., ix-1)` and `b.get(j0, j1, ..., jy-1)` access the same value.

[*Example:* Consider a 1,2-dimensional block `b` with `b.size(2,0) == 2`, `b.size(2,1) == 4`, and `b.size(1,0) == 8`. Figure 6.1 has a 2-dimensional illustration of `b`, which has two rows and four columns. Each value is labelled. Figure 6.2 has a 1-dimensional illustration of `b`, having eight values. Value `g` can be accessed using `b.get(6)` and `b.get(1,2)` since `1 * b.size(2,1) + 2 == 6`.]

- 9 [Note: The ordering of indices and the ordering of actual value storage (if values are stored) need not be the same.]
- 10 [Note: Maps are specified in [view.map].]
- 11 Every modifiable block shall satisfy the requirements in Table 6.2 in addition to the requirements in Table 6.1. In Table 6.2, `b` denotes a modifiable block and `t` denotes a value of type `T`.
- 12 For a block that is x -dimensional, `b.map()` must yield a y -dimensional map where $y \geq x$. [Note: An x -dimensional block will access at most the smallest x dimensions of `b.map()`. The other dimensions are ignored.]

Table 6.2: Modifiable block requirements

expression	return type	assertion/note pre/post-condition
<code>b.put(i_1, i_2, ..., i_x, t)</code>	implementation-defined	post: <code>b.get(i_1, ..., i_x) == t</code>

- 13 For a block explicitly created by a VSIPL++ user, the effect of creation should also include the effect of invoking `increment_count` once.
- 14 [*Note:* For a block explicitly created by a VSIPL++ user, the number of `increment_count` invocations should exceed the number of `decrement_count` invocations if the block is to be used. Such a block may go out of scope or be explicitly destroyed even if the number of `increment_count` invocations exceeds the number of `decrement_count` invocations.]

6.1.1 Allocatable Block

[**view.block.alloc**]

- 1 [*Note:* Some view constructors allocate their underlying blocks. Such constructors must be invoked only if the view's underlying block is allocatable.]
- 2 Blocks may be allocatable or non-allocatable. An *allocatable block* has a specified interface permitting its creation. A *non-allocatable block* does not have the specified interface.
- 3 Let `B` denote an D -dimensional block class containing objects of type `T`, `dom` denote a `Domain<D>`, `map` denote an object with type `B::map_type` or a reference to an object with type `B::map_type`, and `value` denote a value of type `T`. D must be positive. Let `i0, i1, ..., ix` be D `index_types` less than `dom[0].size()`, `dom[1].size()`, ..., `dom[D-1].size()`, respectively.
- 4 `B` is allocatable if both of the following hold.
- `B(dom, map)` is a valid expression constructing a `B` block `b`, `b.get(i0, i1, ..., ix)` is valid, and `b.map()` is a reference to `map`.
 - `B(dom, value, map)` is a valid expression constructing a `B` block, `b.get(i0, i1, ..., ix)` is valid, `b.get(i0, i1, ..., ix) == value`, and `b.map()` is a reference to `map`.
- 5 [*Note:* An allocatable block can be either modifiable or non-modifiable.]

- 6 For a block allocated by a view constructor, the effect of creation should include the effect of invoking `increment_count`.
- 7 [Note: For a block allocated by a view constructor, the number of `increment_count` invocations should meet or exceed the number of `decrement_count` invocations if the block is to be used. When the number of `decrement_count` invocations exceeds, not just equals, `increment_count` invocations, the block should be deallocated.]

6.2 View requirements

[view.view]

- 1 Every *view* is logically a contiguous array with dimension D between one and `VSIP_MAX_DIMENSIONS`, inclusive. The `Domain<D>` associated with a view indicates the set of `Index<D>`s that can be used to access the view. The dimension of a view does not change during its lifetime.
- 2 Views support data-parallel operations, such as the assignment of one view to another or the element-wise addition of two views.
- 3 Every view has an associated block which is responsible for storing or computing the data in the view. More than one view may be associated with the same block.
- 4 Views may be modifiable or non-modifiable. The interface of a *modifiable view* supports changing the view's data; the interface of a *non-modifiable view* does not support changing its data. [Note: The data of a non-modifiable view may be changed by another modifiable view accessing the same data.]
- 5 The type of objects stored in these components must meet the same requirements as specified for blocks.
- 6 Every view shall satisfy the requirements in Table 6.3. In Table 6.3, V denotes a view class of dimension D containing objects of type T with an underlying D -dimensional block of type B , v denotes a value of type V , b denotes a value of type B , d denotes a value of type `dimension_type`, and i_1, i_2, \dots, i_D denote D values of type `index_type`.
- [Note: Many of the view requirements are identical to the container requirements in the ISO C++ standard (ISO14882, [lib.container.requirements]).]
- 7 Two views *overlap* if their domains overlap. Two views *exactly overlap* if their domains exactly overlap.

Table 6.3: View requirements

expression	return type	assertion/note pre/post-condition
<code>V::dim</code>	<code>dimension_type</code>	<code>V::dim == D</code>
<code>V::block_type</code>	<code>B</code>	<code>B is a block</code>
<code>V::value_type</code>	<code>B::value_type</code>	
<code>V::reference_type</code>	<code>B::reference_type</code>	
<code>V::const_reference_type</code>	<code>B::const_reference_type</code>	
<code>V(b)</code>		Constructs <code>V</code> object.
<code>v.block()</code>	<code>B const&</code>	
<code>v.get(i1, i2, ..., iD)</code>	<code>value_type</code>	
<code>v.size()</code>	<code>index_type</code>	post: product of <code>size(d)</code> over all <code>d</code> < <code>D</code>
<code>v.size(d)</code>	<code>index_type</code>	pre: <code>0 <= d < D</code>

- 8 Two views are *element conformant* if their domains are element conformant.
- 9 Two values in separate views *correspond* if the values have corresponding `Index<D>`s.
- 10 Every modifiable view shall satisfy the requirements in Table 6.4 in addition to the requirements in Table 6.3. In Table 6.4, `V` denotes a modifiable view class, `v` denotes an object of type `V`, `t` denotes a value of type `T` and `v2` denotes a view object (possibly of a type other than `V`).

Table 6.4: Modifiable view requirements

expression	return type	assertion/note pre/post-condition
<code>v.put(i1, i2, ..., iD, t)</code>	implementation-defined	post: <code>a.get(i1, ..., iD) == t</code>
<code>v = v2</code>	<code>const V&</code>	pre: <code>v, v2</code> element-conformant

- 11 A *subview* of a view object `v` refers to (possibly a subset of) `v`'s block.
- 12 Subviews use reference semantics. Consider a view `v` and a subview `subv` of `v`, an `Index<D>` position `idx` in `subv`'s domain, and its associated value `val`. Since `subv` is a subview, there is an associated `Index` in `v`'s `Domain<D>`. Changing the value at index position `idx` to a new value `val2` changes the value for both `v` and `subv`.

- 13 [Note: A subview's domain is not required to be a subset of its referenced view's domain. For example, the transpose of a non-square matrix will have one dimension longer than the original matrix.]
- 14 To help blocks implement reference counting, a view's constructor that does not allocate the block must call its block's `increment_count` member function exactly once. A constructor that does allocate its block must not invoke its block's `increment_count` member function. A view's destructor must call its block's `decrement_count` exactly once.
- 15 [Note: Subviews use reference semantics so subviews, including entire copies, of views refer to the same block. Many views, e.g., `Vector`, `Matrix`, and `Tensor`, support modification of values via assignment. Users who are new to C++ should note that any statement beginning with a type specifier and involving an existing object with the same type is almost always a copy, not an assignment, even if an equal sign is present.]

6.2.1 View assignments

[view.view.assign]

- 1 Some views support assignment operations. A single view assignment may modify multiple values in the view. Each individual view value modification is an *individual assignment*. All individual assignments specified by a single assignment are collectively called the *collective assignment*.
- 2 The order of individual assignments within a single collective assignment can affect the collective result. [Example: Consider a collective assignment to a view `v` incorporating two individual assignments `v.put(0, v.get(1))` and `v.put(1, v.get(2))`. Let `v.get(1) == t1` and `v.get(2) == t2`. If the put to index 0 occurs before the put to 1, then, after the collective assignment, `v.get(0) == t1`. If the put to index 1 occurs before the put to 0, then, after the collective assignment, `v.get(0) == t2`.]
- 3 An *order-dependent assignment* is a collective assignment for which the order of individual assignments affects the collective result. An *order-independent assignment* is a collective assignment for which the order of individual assignments does not affect the collective result.
- 4 Assignments must be order-independent. It is the user's responsibility to ensure that assignments are order-independent.

- 5 Given an assignment statement, *order-independent assignment operands* are a left-hand side view v and right-hand side views w_0, \dots such that the assignment is order-independent.

6.3 Dense block

[view.dense]

- 1 A Dense block is a modifiable, allocatable 1-dimensional block or a $1, x$ -dimensional block, for a fixed x , that explicitly stores one value for each Index in its domain.

Header `<vsip/dense.hpp>` synopsis

```
namespace vsip {
    enum user_storage_type {
        no_user_format = false,
        array_format,
        interleaved_format,
        split_format
    };

    template <dimension_type D = 1,
              typename T = VSIP_DEFAULT_VALUE_TYPE,
              typename Order = typename impl::RowMajor<Dim>::type,
              // RowMajor abbreviates the Dth rowD_type type definition.
              typename DenseMap = Map<> >
    class Dense {
    public:
        // compile-time values:
        static dimension_type const dim = D;
        typedef T value_type;
        typedef T& reference_type;
        typedef T const& const_reference_type;
        typedef Order order_type;
        typedef DenseMap map_type;

        // constructors and destructor
        Dense(Domain<D> const&, T const&,
             map_type const& = map_type())
            VSIP_THROW((std::bad_alloc));
        Dense(Domain<D> const&, map_type const& = map_type())
            VSIP_THROW((std::bad_alloc));
        Dense(Domain<D> const&, T*const pointer,
             map_type const& = map_type())
            VSIP_THROW((std::bad_alloc));
```

```

// present only for complex type T = complex<uT>:
Dense(Domain<D> const&, uT*const pointer,
      map_type const& = map_type())
  VSIP_THROW((std::bad_alloc));
// present only for complex T = complex<uT>:
Dense(Domain<D> const&,
      uT*const real_pointer,
      uT*const imag_pointer,
      map_type const& = map_type())
  VSIP_THROW((std::bad_alloc));
~Dense() VSIP_NOTHROW;

// copy constructor and assignment
private:
Dense(Dense const&);
Dense& operator=(Dense const&);

public:
// user data manipulation functions
void admit(bool update = true)
  VSIP_NOTHROW;
void release(bool update = true)
  VSIP_NOTHROW;
void release(bool update, T*& pointer)
  VSIP_NOTHROW;
// present only for complex T = complex<uT>:
void release(bool update,
             uT*& pointer)
  VSIP_NOTHROW;
// present only for complex T = complex<uT>:
void release(bool update,
             uT*& real_pointer,
             uT*& imag_pointer)
  VSIP_NOTHROW;
void find(T*& pointer)
  VSIP_NOTHROW;
// present only for complex T = complex<uT>:
void find(uT*& pointer)
  VSIP_NOTHROW;
// present only for complex T = complex<uT>:
void find(uT*& real_pointer,
          uT*& imag_pointer)
  VSIP_NOTHROW;
void rebind(T*const pointer)
  VSIP_NOTHROW;

```

6.3.1. [VIEW.DENSE.USERVOCAB] USER-SPECIFIED STORAGE ...

```
// present only for complex T = complex<uT>:
void rebind(uT*const pointer)
    VSIP_NOTHROW;
// present only for complex T = complex<uT>:
void rebind(uT*const real_pointer,
            uT*const image_pointer)
    VSIP_NOTHROW;

// value accessors
value_type get(index_type) const VSIP_NOTHROW;
value_type get(index_type, ..., index_type) const VSIP_NOTHROW;
// exactly D index_type parameters; present only if D != 1
unspecified put(index_type, T const&) VSIP_NOTHROW;
unspecified put(index_type, ..., index_type, T const&) VSIP_NOTHROW;
// exactly D index_type parameters; present only if D != 1

// accessors
length_type size() VSIP_NOTHROW;
length_type size(dimension_type Dim, dimension_type d) VSIP_NOTHROW;
void increment_count() VSIP_NOTHROW;
void decrement_count() VSIP_NOTHROW;
map_type const& map() const VSIP_NOTHROW;
enum user_storage_type user_storage() const VSIP_NOTHROW;
bool admitted() const VSIP_NOTHROW;
};
}
```

2 If T is a complex type, let uT be its underlying type.

[*Note:* This header file is usually included in views' header files so direct inclusion is rarely necessary.]

6.3.1 User-specified storage vocabulary

[**view.dense.uservocab**]

1 [*Note:* A VSIPL++ user can rely on the library to allocate a Dense block's storage or can explicitly provide storage. Terminology for the latter, called blocks with user-specified storage, are described here.]

2 [*Note:* A VSIPL++ user may provide one or two arrays for a Dense object to use to store values. Each one-dimensional array of memory is presented as a pointer to a Dense constructor or a subsequent `rebind` call. The block must be admitted before its use by VSIPL++ functions and objects. The pointer(s) and the associated data storage must not be accessed except by VSIPL++ functions and objects while

the block is admitted. Releasing the block ends the admission so the pointer(s) and the associated data storage may be used by non-VSIPL++ functions and objects. Rebinding a block changes its pointer(s). A block's pointer(s) can also be found, i.e., returned to the user. This functionality provides some of the functionality of VSIPL user data arrays.]

- 3 A *block with user-specified storage* is a `Dense` block created by a constructor taking one or two pointers. [Note: The pointer or pointers should indicate storage for the block's data.] A block that is not a block with user-specified storage is a *block without user-specified storage*.
- 4 A block with user-specified storage can be *bound* to a single pointer or to two pointers. The data pointed to by these pointers must be in one of three formats. A single pointer can point to a data array with n values in *array format* if it points to a contiguous array with n values. If T is a complex type, a single pointer can point to a data array in *interleaved format*. The pointer to the underlying type has $2n$ values with even-indexed values represent the real portions of n complex numbers and the odd-indexed values represent the imaginary portions. Two pointers and a complex type T are required to represent n complex values in a *split format*. One pointer points to a data array of length n of the underlying type containing the real portions of the complex numbers, while the other pointer points to the n imaginary portions.
- 5 [Note: `enum user_storage_type` indicates a data array's array, interleaved, or split format. `no_user_format`, indicating a block without user-specified storage, equals the value `false`.]
- 6 The *admit-release* sequence of a block with user-specified storage is the sequence of `admit` and `release` invocations on the block.
- 7 An *admitted block* is a block with user-specified storage such that its admit-release sequence ends with an `admit` invocation. A block without user-specified storage is also considered admitted at all times. A *released block* is a block with user-specified storage such that its admit-release sequence ends with a `release` invocation, or (the sequence) is empty. [Note: A block with user-specified storage that has been newly created is defined to be released.]

6.3.2 Template parameters

[view.dense.template]

- 1 D specifies a dimension that is at least one and at most `VSIP_MAX_DIMENSION`. If D is 1, then the block will be a 1-dimensional block. If D is greater than 1, then the block will be a $1,D$ -dimensional block.

- 2 `T` specifies the type of values stored in the `Dense` object. If $D == 1$, the only specializations which must be supported have `T` the same as `scalar_f`, `scalar_i`, `cscalar_f`, `cscalar_i`, `bool`, `index_type`, `Index<1>`, `Index<2>`, or `Index<3>`. If $D > 1$, the only specializations which must be supported have `T` the same as `scalar_f`, `scalar_i`, `cscalar_f`, `cscalar_i`, or `bool`. An implementation is permitted to prevent instantiation for other choices of `T`.
- 3 `Order` specifies the storage dimension ordering. Its default value is the `row_type` type definition explicitly specifying the first D dimensions, i.e., row-major ordering. [Example: For $D == 2$, the default value is `row2_type`.]

If the implementation does not store the block's values in a one-dimensional ordering, this template parameter is ignored. This template parameter does not affect program correctness or value access.

If an implementation chooses to store a `Dense` block's values in a one-dimensional ordering, the storage dimension ordering indicates how a multi-dimensional block's values are linearized into the ordering.

The one-dimensional ordering must obey the ordering determined by $<$, where $v < w$ indicates v is stored at a memory address less than w . Given a tuple `tuple< $d_0, d_1, \dots, d_{D-1}, \dots$ >`, the value with D -dimensional index $(c_{0,0}, c_{1,0}, \dots, c_{D-1,0}) <$ the value with D -dimensional index $(c_{0,1}, c_{1,1}, \dots, c_{D-1,1})$ if and only if

1. there is some $j \in [0, D)$ such that for all $k \in [0, j)$, $c_{d(k),0} == c_{d(k),1}$ and
2. $c_{d(j),0} < c_{d(j),1}$.

- 4 `DenseMap` must be a map type with a default constructor.

6.3.3 Constructors, copy, assignment, and destructor [view.dense.constructors]

```
Dense (
    Domain<D> const& dom,
    T const&          value,
    map_type const&  map = map_type() )
VSIP_THROW((std::bad_alloc));
```

Effects: Constructs a modifiable `Dense` object containing the map `map` and exactly `dom.size()` values equal to `value`.

Throws: `std::bad_alloc` indicating memory allocation for the returned `Dense` failed.

Postconditions: If `D == 1`, `*this` will have a one-dimensional `Domain<1>` denoted domain with `Index<1>es` `0, ..., dom.size()-1` and `domain.first() == 0`. If `D != 1`, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` `0, ..., dom.size()-1` and `domain1.first() == 0` and a `Domain<D>` `domainD` with, for each `0 <= d < D`, `domainD[d].size() == dom[d].size()`, `domainD[d].stride() == 1`, and `domainD[d].first() == 0`. The object's use count will be one. `this->user_storage() == NO_USER_STORAGE`.

```
Dense (
    Domain<D> const& dom,
    map_type const& map = map_type())
VSIP_THROW((std::bad_alloc));
```

Effects: Constructs a modifiable `Dense` object containing `dom.size()` unspecified values and the map `map`.

Throws: `std::bad_alloc` indicating memory allocation for the returned `Dense` failed.

Postconditions: If `D == 1`, `*this` will have a one-dimensional `Domain<1>` denoted domain with `Index<1>es` containing `0, ..., dom.size()-1`. If `D != 1`, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` containing `0, ..., dom.size()-1` and a `Domain<D>` `domainD` with, for each `0 <= d < D`, `domainD[d].size() == dom[d].size()`, `domainD[d].stride() == 1`, and `domainD[d].first() == 0`. The object's use count will be one. `this->user_storage() == NO_USER_STORAGE`.

Note: The block's values are unspecified.

```
Dense (
    Domain<D> const& dom,
    T*const          pointer,
    map_type const& map = map_type())
VSIP_THROW((std::bad_alloc));
```

Requires: For all `i` such that `0 <= i && i < dom.size()`, `pointer[i] = T()` must be a valid C++ expression.

Effects: Constructs a modifiable `Dense` object containing the map `map`. If `map` indicates a distributed block, this subblock contains `dom.size()` unspecified values.

Throws: `std::bad_alloc` indicating memory allocation for the returned `Dense` failed.

Postconditions: If `D == 1`, `*this` will have a one-dimensional `Domain<1>` denoted domain with `Index<1>es` containing `0, ..., dom.size()-1`. If `D != 1`, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` containing `0, ..., dom.size()-1` and a `Domain<D>` `domainD` with, for each `0 <= d < D`, `domainD[d].size() == dom[d].size()`, `domainD[d].stride() == 1`, and `domainD[d].first() == 0`. The object's use count will be one. `this->user_storage() == array_format`.

Note: The block's values are unspecified. This block's values can only be accessed after an `admit` call and before its corresponding `release` call. When the block is admitted, the `pointer[i]` values listed above may be modified by the block.

A VSIP++ Library implemented using a VSIPL implementation cannot provide this constructor for complex type `T` in a portable way because the C++ and VSIP++ standards do not specify a particular complex number representation. Thus, it cannot be guaranteed to match VSIPL functionality.

Dense (

```
    Domain<D> const& dom,
    uT*const        pointer,
    map_type const& map = map_type()
VSIP_THROW((std::bad_alloc));
```

Requires: `T` must be a complex type. For all `i` such that `0 <= i && i < 2*dom.size()`, `pointer[i] = uT()` must be a valid C++ expression.

Effects: Constructs a modifiable `Dense` object containing the map `map`. If `map` indicates a distributed block, this subblock contains `dom.size()` unspecified values.

Throws: `std::bad_alloc` indicating memory allocation for the returned `Dense` failed.

Postconditions: If `D == 1`, `*this` will have a one-dimensional `Domain<1>` denoted domain with `Index<1>es` containing `0, ..., dom.size()-1`. If `D != 1`, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` containing `0, ..., dom.size()-1` and a `Domain<D>` `domainD` with, for each `0 <= d < D`, `domainD[d].size() == dom[d].size()`, `domainD[d].stride() == 1`, and `domainD[d].first() == 0`. The object's use count will be one. `this->user_storage() == interleaved_format`.

Note: The block's values are unspecified. This block's values can only be accessed after an `admit` call and before its corresponding `release` call. When the block is admitted, the `pointer[i]` values listed above may be modified by the block.

```
Dense (
    Domain<D> const& dom,
    uT*const          real_pointer,
    uT*const          imag_pointer,
    map_type const& map = map_type())
VSIP_THROW((std::bad_alloc));
```

Requires: T must be a complex type. For all i such that $0 \leq i \ \&\& \ i < \text{dom.size}()$, `real_pointer[i] = uT()` and `imag_pointer[i] = uT()` must be valid C++ expressions.

Effects: Constructs a modifiable `Dense` object containing the map `map`. If `map` indicates a distributed block, this subblock contains `dom.size()` unspecified values.

Throws: `std::bad_alloc` indicating memory allocation for the returned `Dense` failed.

Postconditions: If $D == 1$, `*this` will have a one-dimensional `Domain<1>` denoted domain with `Index<1>es` containing $0, \dots, \text{dom.size}()-1$. If $D \neq 1$, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` containing $0, \dots, \text{dom.size}()-1$ and a `Domain<D>` `domainD` with, for each $0 \leq d < D$, `domainD[d].size() == dom[d].size()`, `domainD[d].stride() == 1`, and `domainD[d].first() == 0`. The object's use count will be one. `this->user_storage() == split_format`.

Note: The block's values are unspecified. This block's values can only be accessed after an `admit` call and before its corresponding `release` call. When the block is admitted, the `real_pointer[i]` and `imag_pointer[i]` values listed above may be modified by the block.

```
~ Dense () VSIP_NOTHROW;
```

Effects: Destroys the `Dense` object.

Postconditions: It should no longer be used.

Note: The object is destroyed regardless of its reference count.

*6.3.4 User-specified storage***[view.dense.userdata]**

```
void  
admit(bool update = true) VSIP_NOTHROW;
```

Effects: If `*this` is not a block with user-specified storage or `*this` is an admitted block with user-specified storage, there is no effect.

Otherwise, the block is admitted so that its data can be used by VSIPL++ functions and objects. If `update == true`, the values of `*this` are updated. That is, assuming `this->user_storage() == array_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == pointer[i]`, where `pointer` is the value returned by `this->find`. Assuming `this->user_storage() == interleaved_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == complex(pointer[2i], pointer[2i+1])`, where `pointer` is the value returned by `this->find`. Assuming `this->user_storage() == split_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == complex(real_pointer[i], imag_pointer[i])`, where `real_pointer` and `imag_pointer` are the values returned by `this->find`.

If `update == false`, the result of `this->get(i)` for all `0 <= i && i < this->size()` is unspecified.

Note: Invoking `admit` on an admitted block is permitted. The intent of using a false update flag is that, if the data in the user-specified storage is not needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

```
void  
release(bool update = false) VSIP_NOTHROW;
```

Effects: If `*this` is not a block with user-specified storage or `*this` is a released block with user-specified storage, there is no effect. Otherwise, the block is released so that its data cannot be used by VSIPL++ functions and objects.

If `update == true`, the values in the user-specified storage are updated. Assuming `this->user_storage() == array_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == pointer[i]`, where `pointer` is the value returned by `this->find`.

Assuming `this->user_storage() == interleaved_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == complex(pointer[2i], pointer[2i+1])`, where `pointer` is the value returned by `this->find`. Assuming `this->user_storage() == split_format`, for all `i` such that `0 <= i && i < this->size()`, `this->get(i) == complex(real_pointer[i], imag_pointer[i])`, where `real_pointer` and `imag_pointer` are the values returned by `this->find`.

If `update == false`, the values in the user-specified storage are unspecified. Assuming `this->user_storage() == array_format` and `pointer` is the value returned by `this->find`, for all `0 <= i && i < this->size()`, `pointer[i]` is unspecified. Assuming `this->user_storage() == interleaved_format` and `pointer` is the value returned by `this->find`, for all `0 <= i && i < 2*this->size()`, `pointer[i]` is unspecified. Assuming `this->user_storage() == split_format` and `real_pointer` and `imag_pointer` are the values returned by `this->find`, for all `0 <= i && i < this->size()`, `real_pointer[i]` and `imag_pointer[i]` are unspecified.

Note: Invoking `release` on a released block is permitted. The intent of using a `false` `update` flag is that, if the data in the user-specified storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

`void`

release(`bool update`, `T*& pointer`) `VSIP_NOTHROW`;

Requires: `*this` must not be a block with user-specified storage or it must be a block with user-specified storage such that `this->user_storage() == array_format`.

Effects: If `*this` is not a block with user-specified storage, `pointer` is assigned `NULL`, but there are no other effects. If `*this` is a released block with user-specified storage, `pointer` is assigned the value returned by `this->find`, but there are no other effects.

Otherwise, the block is released so that its data may not be used by VSIP++ functions and objects. `pointer` is assigned the value returned by `this->find`. If `update == true`, the values in the user-specified storage are updated. For all `i` such that `0 <= i && i < this->size()`,

`this->get(i) == pointer[i]`. If `update == false`, the values in the user-specified storage are unspecified.

Note: Invoking `release` on a released block is permitted. The intent of using a false `update` flag is that, if the data in the user-specified storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

```
void  
release(bool update, uT*& pointer) VSIP_NOTHROW;
```

Requires: `*this` must not be a block with user-specified storage or it must be a block with user-specified storage such that `this->user_storage() == interleaved_format`. `T` must be a complex type.

Effects: If `*this` is not a block with user-specified storage, `pointer` is assigned `NULL`, but there are no other effects. If `*this` is a released block with user-specified storage, `pointer` is assigned the value returned by `this->find`, but there are no other effects.

Otherwise, the block is released so that its data may not be used by VSIP++ functions and objects. `pointer` is assigned the value returned by `this->find`. If `update == true`, the values in the user-specified storage are updated. For all `i` such that `0 <= i && i < this->size()`, `this->get(i) == complex(pointer[2i], pointer[2i+1])`. If `update == false`, the values in the user-specified storage are unspecified.

Note: Invoking `release` on a released block is permitted. The intent of using a false `update` flag is that, if the data in the user-specified storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

```
void  
release(  
    bool update,  
    uT*& real_pointer,  
    uT*& imag_pointer)  
VSIP_NOTHROW;
```

Requires: `*this` must not be a block with user-specified storage or it must be a block with user-specified storage such that `this->user_storage()` equals `interleaved_format` or `split_format`. `T` must be a complex type.

Effects: If `*this` is not a block with user-specified storage, `real_pointer` and `imag_pointer` are assigned `NULL`, but there are no other effects. If `*this` is a released block with user-specified storage, `real_pointer` and `imag_pointer` are assigned the values returned by `this->find`, but there are no other effects. Otherwise, the block is released so that its data may not be used by VSIP++ functions and objects. `real_pointer` and `imag_pointer` are assigned the values returned by `this->find`.

If `update == true`, the values in the user-specified storage are updated. Assuming `this->user_storage() == interleaved_format`, `this->get(i) == complex(real_pointer[2i], real_pointer[2i+1])` for all `i` such that `0 <= i && i < this->size()`. Assuming `this->user_storage() == split_format`, `this->get(i) == complex(real_pointer[i], imag_pointer[i])` for all `i` such that `0 <= i && i < this->size()`.

If `update == false`, the values in the user-specified storage are unspecified. Assuming `this->user_storage() == interleaved_format`, for all `0 <= i && i < 2 * this->size()`, `real_pointer[i]` is unspecified. Assuming `this->user_storage() == split_format`, for all `0 <= i && i < this->size()`, `real_pointer[i]` and `imag_pointer[i]` are unspecified.

Note: Invoking `release` on a released block is permitted. The intent of using a `false` `update` flag is that, if the data in the user-specified storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

```
void
find(T*& pointer) VSIP_NOTHROW;
```

Requires: `*this` must not be a block with user-specified storage or it must be a block with `this->user_storage() == array_format`.

Effects: If `*this` is not a block with user-specified storage, `pointer` is assigned `NULL`. If `*this` is an admitted block with user-specified storage, `pointer` is assigned `NULL`. If `!this->admitted()`, `pointer` is assigned the latest pointer bound to the block.

```
void
find(uT*& pointer) VSIP_NOTHROW;
```

Requires: `*this` must not be a block with user-specified storage or it must be a block with `this->user_storage() == interleaved_format`. `T` must be a complex type.

Effects: If `*this` is not a block with user-specified storage, `pointer` is assigned `NULL`. If `*this` is an admitted block with user-specified storage, `pointer` is assigned `NULL`. If `!this->admitted()`, `pointer` is assigned the latest pointer bound to the block.

```
void  
find(uT*& real_pointer, uT*& imag_pointer)  
VSIP_NOTHROW;
```

Requires: `*this` must not be a block with user-specified storage or it must be a block with `this->user_storage()` equaling either `interleaved_format` or `split_format`. `T` must be a complex type.

Effects: If `*this` is not a block with user-specified storage, `real_pointer` and `imag_pointer` are assigned `NULL`. If `*this` is an admitted block with user-specified storage, `real_pointer` and `imag_pointer` are assigned `NULL`.

Otherwise `!this->admitted()`. If `this->user_storage() == interleaved_format`, `real_pointer` is assigned the latest pointer bound to the block, and `imag_pointer` is assigned `NULL`. If `this->user_storage() == split_format`, `real_pointer` and `imag_pointer` are assigned to the latest pointers bound to the block.

```
void  
rebind(T*const pointer) VSIP_NOTHROW;
```

Requires: `!this->admitted() && this->user_storage() == array_format`. For all `i` such that `0 <= i && i < dom.size()`, `pointer[i] = T()` must be a valid C++ expression.

Effects: Replaces the block's user-specified storage pointer with `pointer`.

```
void  
rebind(uT*const pointer) VSIP_NOTHROW;
```

Requires: `!this->admitted()`. `this->user_storage()` equals `interleaved_format` or `split_format`. `T` must be a complex type. For all `i` such that `0 <= i && i < 2 * dom.size()`, `pointer[i] = T()` must be a valid C++ expression.

Effects: Replaces the block's user-specified storage pointer with `pointer`.

Postconditions: `this->user_storage() == interleaved_format`.

```
void
rebind(uT*const real_pointer, uT*const imag_pointer)
VSIP_NOTHROW;
```

Requires: `!this->admitted()`. `this->user_storage()` equals `interleaved_format` or `split_format`. `T` must be a complex type. For all `i` such that `0 <= i && i < dom.size()`, `real_pointer[i] = T()` and `imag_pointer[i] = T()` must be valid C++ expressions.

Effects: If `*this` is not a block with user-specified storage, this function has no effect. Otherwise, replaces the block's user-specified storage pointers with `real_pointer` and `imag_pointer`.

Postconditions: `this->user_storage() == split_format`.

6.3.5 Value Accessors

[**view.dense.valaccess**]

[*Note:* [`view.dense`] requires `get` and `put` functions taking one `index_type` operand and, if `D != 1`, taking exactly `D` operands. The restrictions for `get` and `put` follow from the block requirements in [`view.block`]. Only additional restrictions occur here.]

- 1 A Dense block must be admitted to use `get` or `put`.

```
value_type
get(index_type, ..., index_type)
const VSIP_NOTHROW;
```

Requires: This member function is present only if `D != 1`.

unspecified

```
put(
    index_type, ..., index_type,
    T const&)
VSIP_NOTHROW;
```

Requires: This member function is present only if `D != 1`.

6.3.6 Accessors

[view.dense.accessors]

```
length_type  
size() const VSIP_NOTHROW;
```

Returns: The number of values in the block.

```
length_type  
size(dimension_type Dim, dimension_type d)  
VSIP_NOTHROW;
```

Requires: `Dim == 1` or `Dim == D.0` `<= d` && `d < Dim`.

Effects: The number of values in dimension `d` of the block when viewed as a `Dim`-dimensional block.

```
void  
increment_count() VSIP_NOTHROW;
```

Effects: Increase the object's use count.

```
void  
decrement_count() VSIP_NOTHROW;
```

Effects: Decrease the object's use count. If the count becomes zero, the block deallocates itself.

```
map_type const&  
map() const VSIP_NOTHROW;
```

Returns: The block's map as given to the constructor.

```
enum user_storage_type  
user_storage()  
const VSIP_NOTHROW;
```

Returns: `no_user_format` if `*this` is a block without user storage. `array_format` if `*this` is a user-specified block bound to an array with array format. `interleaved_format` if `*this` is a user-specified block bound to a complex array with interleaved format. `split_format` if `*this` is a user-specified block bound to a complex array with split format.

```
bool  
admitted() const VSIP_NOTHROW;
```

Returns: `true` if and only if `*this` is admitted.

6.4 Vector

[view.vector]

- 1 The `const_Vector` and `Vector` classes are views implementing the mathematical idea of vectors, i.e., one-dimensional storage and access to values. A `const_Vector` view is not modifiable, but a `Vector` view is modifiable.
- 2 The interfaces for `const_Vector` and `Vector` are similar so they are simultaneously specified except where noted. For these, `Vector` indicates both `const_Vector` and `Vector`. The term “vector” refers to a `const_Vector` or `Vector` object.

Header <vsip/vector.hpp> synopsis

```
namespace vsip {
    template <typename T = VSIP_DEFAULT_VALUE_TYPE,
              typename Block = Dense<1, T> >
        class const_Vector;

    template <typename T = VSIP_DEFAULT_VALUE_TYPE,
              typename Block = Dense<1, T> >
        class Vector;

    template <typename T = VSIP_DEFAULT_VALUE_TYPE,
              typename Block = Dense<1, T> >
        class const_Vector {
        public:
            // compile-time values
            static dimension_type const dim = 1;
            typedef Block block_type;
            typedef typename block_type::value_type value_type;
            typedef typename block_type::reference_type reference_type;
            typedef typename block_type::const_reference_type
                const_reference_type;

            // subview types
            // [view.vector.subview_types]
            typedef const_Vector<T, unspecified> subview_type;
            typedef subview_type const_subview_type;
            // present only for complex type T = complex<Tp>:
            typedef const_Vector<Tp, unspecified> realview_type;
            typedef realview_type const_realview_type;
            typedef const_Vector<Tp, unspecified> imagview_type;
            typedef imagview_type const_imagview_type;
        };
};
```

```
// constructors, copy, and destructor
// [view.vector.constructors]
const_Vector(length_type, T const& value);
explicit const_Vector(length_type);
const_Vector(Block&) VSIP_NOTHROW;
const_Vector(const_Vector const&) VSIP_NOTHROW;
const_Vector(Vector const&) VSIP_NOTHROW;
~const_Vector() VSIP_NOTHROW;

// value accessors
// [view.vector.valaccess]
value_type get(index_type) const VSIP_NOTHROW;
const_reference_type operator()(index_type) const VSIP_NOTHROW;

// subview accessors
// [view.vector.subviews]
const_subview_type
    get(Domain<1> const&) const VSIP_THROW((std::bad_alloc));
const_subview_type
    operator()(Domain<1> const&) const VSIP_THROW((std::bad_alloc));
// present only for complex type T:
const_realview_type
    real() const VSIP_THROW((std::bad_alloc));
const_imagview_type
    imag() const VSIP_THROW((std::bad_alloc));

// accessors
// [view.vector.accessors]
block_type const& block() const VSIP_NOTHROW;
length_type size() const VSIP_NOTHROW;
length_type size(dimension_type) const VSIP_NOTHROW;
length_type length() const VSIP_NOTHROW;
};

template <typename T = VSIP_DEFAULT_VALUE_TYPE,
          typename Block = Dense<1, T> >
class Vector {
public:
    // compile-time values
    static dimension_type const dim = 1;
    typedef Block block_type;
    typedef typename block_type::value_type value_type;
    typedef typename block_type::reference_type reference_type;
    typedef typename block_type::const_reference_type
        const_reference_type;
```

```

// subview types
// [view.vector.subview_types]
typedef Vector<T, unspecified> subview_type;
typedef const_Vector<T, unspecified> const_subview_type;
// present only for complex type T = complex<Tp>:
typedef Vector<Tp, unspecified> realview_type;
typedef const_Vector<Tp, unspecified> const_realview_type;
typedef Vector<Tp, unspecified> imagview_type;
typedef const_Vector<Tp, unspecified> const_imagview_type;

// constructors, copy, assignment, and destructor
// [view.vector.constructors]
Vector(length_type, T const& value);
explicit Vector(length_type);
Vector(Block&) VSIP_NOTHROW;
Vector(Vector const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Vector(Vector<T0, Block0> const&);
template <typename T0, typename Block0>
Vector& operator=(const_Vector<T0, Block0>)
    VSIP_NOTHROW;
Vector& operator=(const_reference_type) VSIP_NOTHROW;
template <typename T0>
Vector& operator=(T0 const&) VSIP_NOTHROW;
~Vector() VSIP_NOTHROW;

// assignment operators
// [view.vector.assign]
template <typename T0>
Vector& operator+=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Vector& operator+=(const_Vector<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Vector& operator+=(Vector<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Vector& operator-=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Vector& operator-=(const_Vector<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Vector& operator-=(Vector<T0, Block0>)

```

```
|     VSIP_NOTHROW;

    template <typename T0>
    Vector& operator*=(T0 const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Vector& operator*=(const_Vector<T0, Block0>)
        VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Vector& operator*=(Vector<T0, Block0>)
|       VSIP_NOTHROW;

    template <typename T0>
    Vector& operator/=(T0 const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Vector& operator/=(const_Vector<T0, Block0>)
        VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Vector& operator/=(Vector<T0, Block0>)
|       VSIP_NOTHROW;

    template <typename T0>
    Vector& operator&=(T0 const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Vector& operator&=(const_Vector<T0, Block0>)
        VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Vector& operator&=(Vector<T0, Block0>)
|       VSIP_NOTHROW;

    template <typename T0>
    Vector& operator|=(T0 const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Vector& operator|=(const_Vector<T0, Block0>)
        VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Vector& operator|=(Vector<T0, Block0>)
|       VSIP_NOTHROW;

    template <typename T0>
    Vector& operator^=(T0 const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Vector& operator^=(const_Vector<T0, Block0>)
        VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Vector& operator^=(Vector<T0, Block0>)
```

```

|     VSIP_NOTHROW;

    // value accessors
    // [view.vector.valaccess]
    value_type get(index_type) const VSIP_NOTHROW;
    unspecified put(index_type, T const&) VSIP_NOTHROW;
    reference_type operator()(index_type) VSIP_NOTHROW;
    const_reference_type operator()(index_type) const VSIP_NOTHROW;

    // subview accessors
    // [view.vector.subviews]
    const_subview_type
        get(Domain<1> const&) const VSIP_THROW((std::bad_alloc));
    subview_type
        operator()(Domain<1> const&) VSIP_THROW((std::bad_alloc));
    const_subview_type
        operator()(Domain<1> const&) const VSIP_THROW((std::bad_alloc));
    // present only for complex type T:
    realview_type real() VSIP_THROW((std::bad_alloc));
    const_realview_type
        real() const VSIP_THROW((std::bad_alloc));
    imagview_type imag() VSIP_THROW((std::bad_alloc));
    const_imagview_type
        imag() const VSIP_THROW((std::bad_alloc));

    // accessors
    // [view.vector.accessors]
    block_type& block() const VSIP_NOTHROW;
    length_type size() const VSIP_NOTHROW;
    length_type size(dimension_type) const VSIP_NOTHROW;
    length_type length() const VSIP_NOTHROW;
};

// a specialization
template <typename Dim0 = Block, ...,
          typename Dimn = Block>
    // exactly VSIP_MAX_DIMENSION template parameters
    class Vector<T, Dense<1, T, Map<Dim0, ..., Dimn> > > {
public:

    // ... All members are the same as for Vector<T,
    // Block> except Vector's first two constructors are
    // replaced by:

    Vector(length_type, T const& value,

```

6.4.1. [VIEW.VECTOR.TEMPLATE] TEMPLATE PARAMETERS

```
        Dim0 const& = Dim0(), ...,
        Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
explicit Vector(length_type, Dim0 const& = Dim0(), ...,
    Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Vector(length_type, T const& value, const_Vector<processor_type>,
    Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Vector(length_type, const_Vector<processor_type>,
    Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Vector(length_type, T const& value, processor_type const*,
    Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Vector(length_type, processor_type const*,
    Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
};

// view conversion
template <template <typename, typename> class View,
    typename T, typename Block>
class ViewConversion;
}
```

6.4.1 *Template parameters*

[view.vector.template]

1 T specifies the type of values stored in the *Vector* object which has an associated block with type *Block* for storing the values. The only specializations which must be supported have T the same as *scalar_f*, *scalar_i*, *cscalar_f*, *cscalar_i*, *bool*, *index_type*, *Index<1>*, *Index<2>*, or *Index<3>*. An implementation is permitted to prevent instantiation for other choices of T.

2 *Block*

Requires: T must be *Block::value_type*. *Block* must be a one-dimensional block.

Note: *Block* need not be modifiable.

6.4.2 *Subview types*

[view.vector.subview_types]

- 1 `SubviewType` specifies the type of a subview of a *Vector*. The type is a *Vector* with value type `T` and an unspecified block type.
- 2 `ConstSubviewType` specifies the type of a non-modifiable subview of a *Vector*. The type is a `const_Vector` with value type `T` and an unspecified block type.
- 3 `RealViewType` specifies the type of a subview of a *Vector* containing only the real parts of the *Vector*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Vector* with value type `Tp` and an unspecified block type.
- 4 `ConstRealViewType` specifies the type of a non-modifiable subview of a *Vector* containing only the real parts of the *Vector*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Vector` with value type `Tp` and an unspecified block type.
- 5 `ImagViewType` specifies the type of a subview of a *Vector* containing only the imaginary parts of the *Vector*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Vector* with value type `Tp` and an unspecified block type.
- 6 `ConstImagViewType` specifies the type of a non-modifiable subview of a *Vector* containing only the imaginary parts of the *Vector*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Vector` with value type `Tp` and an unspecified block type.

6.4.3 *Constructors, copy, assignment, and destructor* [view.vector.constructors]

```
Vector(length_type len, T const& value);
```

Requires: `len > 0`. `Block` must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Vector(Block(Domain<1>(len), value, Block::map_type()))`.

Notes: `Blocks` are created with the effect of `increment_count`, `Vector` does not invoke `increment_count` again for blocks it allocates.

```
Vector(length_type len);
```

Requires: `len > 0`. `Block` must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Vector(Block(Domain<1>(len), Block::map_type()))`.

Notes: Blocks are created with the effect of `increment_count`, `Vector` does not invoke `increment_count` again for blocks it allocates.

```
Vector(Block& block) VSIP_NOTHROW;
```

Requires: 1-dimensional modifiable `block`.

Effects: Constructs a `Vector v` with associated block `block`. `v.size(0) == block.size(1,0)`.

```
const_Vector(Block& block) VSIP_NOTHROW;
```

Requires: 1-dimensional `block`.

Effects: Constructs a `const_Vector v` with associated block `block`. `v.size(0) == block.size(1,0)`.

```
Vector(Vector const& v) VSIP_NOTHROW;
```

Effects: Constructs a subview `Vector` object of `v` such that its domain is the same as `v`'s domain.

Note: `*this` and `v` are functionally equivalent.

```
const_Vector(Vector const& v) VSIP_NOTHROW;
```

Effects: Constructs a subview `const_Vector` object of `v` such that its domain is the same as `v`'s domain.

```
template <typename T0,
          typename Block0>
Vector(Vector<T0, Block0> const& v);
```

Requires: `Block` must be allocatable. The only specializations which must be supported are for `T0` the same as `T`. An implementation is permitted to prevent instantiation for other choices of `T0`. Type `T0` must be assignable to `T`.

Effects: Equivalent to `Vector(Block(Domain<1>(v.size()), Block::map_type())); *this = v;`

```
template <typename T0,
          typename Block0>
Vector<T,Block>&
operator=(const_Vector<T0,Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. The only specializations which must be supported are for `T0` the same as `T`. An implementation is permitted to prevent instantiation for other choices of `T0`. Type `T0` must be assignable to `T`.

Returns: `*this`.

Postconditions: For all index positions `idx` in the domain, `v.get(idx) == this->get(idx)`.

```
Vector<T,Block>&
operator=(const_reference_type val)
VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation if `T` is not `scalar_f`, `scalar_i`, or `cscalar_f`.

Returns: `*this`.

Postconditions: For all index positions `idx` in the domain, `this->get(idx) == val`.

Note: This function corresponds to VSIPL functions `vsip_vfill_i`, `vsip_vfill_f`, and `vsip_cvfill_f`.

```
template <typename T0>
Vector<T,Block>&
operator=(T0 const& val) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation for `T0` different than `T`. An implementation is permitted to prevent instantiation if `T` is not `scalar_f`, `scalar_i`, or `cscalar_f`. Type `T0` must be assignable to `T`.

Returns: `*this`.

Postconditions: For all index positions `idx` in the domain, `this->get(idx) == val`.

```
~ Vector() VSIP_NOTHROW;
```

Effects: If this object is the only one using its block, the block is deleted. Otherwise, its block's use count is decremented by one. Regardless, the object is deallocated.

6.4.4 Assignment operators

[view.vector.assign]

1 Vector, but not const_Vector, has assignment operators.

```
template <typename T0>
Vector&
operator+=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T the same as `scalar_f` or `scalar_i`, the only specializations which must be supported are for T0 the same as T. For T the same as `cscalar_f`, the only specializations which must be supported are for T0 the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations. T0 must be assignable to T.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) + v)`.

Returns: `*this`.

Note: This function corresponds to some of the functionality of VSIPL functions `vsip_svadd_i`, `vsip_svadd_f`, `vsip_rscvadd_f`, and `vsip_csvadd_f`.

```
template <typename T0,
          typename Block0>
Vector&
operator+=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Vector&
operator+=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. The only specializations which must be supported are for T0 the same as T. An implementation is permitted

to prevent instantiation for other choices of T_0 . An implementation is permitted to prevent instantiation for T not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) + v.get(idx))`.

Returns: `*this`.

```
template <typename T0>
Vector&
operator+=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T the same as `scalar_f` or `scalar_i`, the only specializations which must be supported are for T_0 the same as T . For T the same as `cscalar_f`, the only specializations which must be supported are for T_0 the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) - v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
Vector&
operator+=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Vector&
operator-=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. The only specializations which must be supported are for T_0 the same as T . An implementation is permitted to prevent instantiation for other choices of T_0 . An implementation is permitted to prevent instantiation for T not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) - v.get(idx))`.

Returns: `*this`.

```
template <typename T0>
Vector&
operator*=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T equal to *scalar_f*, the only specialization which must be supported is for $T0$ the same as T . For T the same as *cscalar_f*, the only specializations which must be supported are for $T0$ the same as *scalar_f* or *cscalar_f*. An implementation is permitted to prevent other instantiations. $T0$ must be assignable to T .

Effects: For all index positions *idx* in **this*'s domain, *this->put(idx, this->get(idx) * v)*.

Returns: **this*.

```
template <typename T0,
          typename Block0>
Vector&
operator*=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Vector&
operator*=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: *v* must be element-conformant with **this*. **this* and *v* must be order-independent assignment operands. The only specializations which must be supported are for $T0$ the same as T . An implementation is permitted to prevent instantiation for other choices of $T0$. An implementation is permitted to prevent instantiation for T not the same as *scalar_i*, *scalar_f*, or *cscalar_f*. $T0$ must be assignable to T .

Effects: For all index positions *idx* in **this*'s domain, *this->put(idx, this->get(idx) * v.get(idx))*.

Returns: **this*.

```
template <typename T0>
Vector&
operator/=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T equal to *scalar_f*, the only specialization which must be supported is for $T0$ the same as T . For T the same as *cscalar_f*, the only

specialization which must be supported is for T_0 the same as `scalar_f`. An implementation is permitted to prevent other instantiations. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) / v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
Vector&
operator/=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Vector&
operator/=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. The only specializations which must be supported are for T_0 the same as T . An implementation is permitted to prevent instantiation for other choices of T_0 . An implementation is permitted to prevent instantiation for T not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) / v.get(idx))`.

Returns: `*this`.

```
template <typename T0>
Vector&
operator&=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. T_0 must be assignable to T .

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) & v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
```

```
Vector&  
operator&=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,  
          typename Block0>
```

```
Vector&  
operator&=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) & v.get(idx))`.

Returns: `*this`.

```
template <typename T0>  
Vector&  
operator|=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. `T0` must be assignable to `T`.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) | v)`.

Returns: `*this`.

```
template <typename T0,  
          typename Block0>  
Vector&  
operator|=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,  
          typename Block0>  
Vector&  
operator|=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) | v.get(idx))`.

Returns: `*this`.

```
template <typename T0>
Vector&
operator^=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. `T0` must be assignable to `T`.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) ^ v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
Vector&
operator^=(const_Vector<T0, Block0> v) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Vector&
operator^=(Vector<T0, Block0> v) VSIP_NOTHROW;
```

Requires: `v` must be element-conformant with `*this`. `*this` and `v` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions `idx` in `*this`'s domain, `this->put(idx, this->get(idx) ^ v.get(idx))`.

Returns: `*this`.

6.4.5 Value accessors

[view.vector.valaccess]

[*Note:* The restrictions for `get` and `put` follow from the view requirements in [view.view]. Only additional restrictions occur here.]

```
reference_type
operator () (index_type idx) VSIP_NOTHROW;
```

Requires: This value accessor is provided only by `Vector`, not `const_Vector`. `idx < size()`. An implementation should provide this accessor but is not required to do so.

Returns: The value at the index position `idx`.

Note: A VSIP++ Library implemented using a VSIP implementation cannot provide this accessor because VSIP does not provide such functionality.

```
const_reference_type  
operator () (index_type idx)  
const VSIP_NOTHROW;
```

Requires: `idx < size()`. An implementation should provide this accessor but is not required to do so.

Returns: Value at the index position `idx`.

Note: A VSIP++ Library implemented using a VSIP implementation can provide this accessor but not its non-const version.

1 [*Note:* `const_Vector` and `Vector` element access uses parentheses, not brackets, since parentheses, not brackets, are used for multi-dimensional views.
]

6.4.6 Subviews

[**view.vector.subviews**]

1 These functions return subviews of a `const_Vector` or `Vector` object.

```
const_subview_type  
get (Domain<1> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: `d` is a subset of the `Vector`'s domain.

Returns: A subview object of `*this` with domain `Domain<1>(d.size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
subview_type  
operator () (Domain<1> const& d) VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Vector` object. `d` is a subset of the `Vector`'s domain.

Returns: A subview `Vector` object of `*this` with domain `Domain<1>(d.size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_subview_type
operator () (Domain<1> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: *d* is a subset of the *Vector*'s domain.

Returns: A subview object of **this* with domain `Domain<1>(d.size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_realview_type
real () const VSIP_THROW((std::bad_alloc));
```

Requires: *T* must be a complex type.

Returns: A subview `const_Vector` object of **this* with the same domain but accessing only the real parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
realview_type
real () VSIP_THROW((std::bad_alloc));
```

Requires: *T* must be a complex type.

Returns: A subview *Vector* object of **this* with the same domain but accessing only the real parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_imagview_type
imag () const VSIP_THROW((std::bad_alloc));
```

Requires: *T* must be a complex type.

Returns: A subview `const_Vector` object of **this* with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
imagview_type
imag () VSIP_THROW((std::bad_alloc));
```

Requires: *T* must be a complex type.

Returns: A subview *Vector* object of **this* with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

*6.4.7 Accessors***[view.vector.accessors]**

```
block_type const&  
block() const VSIP_NOTHROW;  
block_type&  
block() const VSIP_NOTHROW;
```

Returns: The vector's underlying block. `const_Vector::block()` returns a `block_type const&` while `Vector::block()` returns a `block_type &`.

```
length_type  
size() const VSIP_NOTHROW;
```

Returns: The number of values in the vector.

```
length_type  
size(dimension_type) const VSIP_NOTHROW;
```

Returns: `this->size()`.

```
length_type  
length() const VSIP_NOTHROW;
```

Returns: `this->size()`.

*6.4.8 Vector type conversion***[view.vector.convert]**

- 1 [*Note:* The class `ViewConversion` converts between constant and non-constant view classes.]
- 2 `ViewConversion<Vector, T, Block>::const_view_type` equals `const_Vector<T, Block>` and `ViewConversion<Vector, T, Block>::view_type` equals `Vector<T, Block>`.

*6.5 Matrix***[view.matrix]**

- 1 The `const_Matrix` and `Matrix` classes are views implementing the mathematical idea of matrices, i.e., two-dimensional storage and access to values. A `const_Matrix` view is not modifiable, but a `Matrix` view is modifiable.

- 2 The interfaces for `const_Matrix` and `Matrix` are similar so they are simultaneously specified except where noted. For these, *Matrix* indicates both `const_Matrix` and `Matrix`. The term “matrix” refers to a `const_Matrix` or `Matrix` object.

Header <vsip/matrix.hpp> synopsis

```
namespace vsip {
  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<2, T> >
    class const_Matrix;

  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<2, T> >
    class Matrix;

  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<2, T> >
    class const_Matrix {
  public:
    // compile-time values
    static dimension_type const dim = 2;
    typedef Block block_type;
    typedef typename block_type::value_type value_type;
    typedef typename block_type::reference_type reference_type;
    typedef typename block_type::const_reference_type
      const_reference_type;

    // subview types
    // [view.matrix.subview_types]
    typedef const_Matrix<T, unspecified> subview_type;
    typedef subview_type const_subview_type;
    typedef const_Vector<T, unspecified> col_type;
    typedef col_type const_col_type;
    typedef const_Vector<T, unspecified> diag_type;
    typedef diag_type const_diag_type;
    typedef const_Vector<T, unspecified> row_type;
    typedef row_type const_row_type;
    typedef const_Matrix<T, unspecified> transpose_type;
    typedef transpose_type const_transpose_type;
    // present only for complex type T = complex<Tp>:
    typedef const_Matrix<Tp, unspecified> realview_type;
    typedef realview_type const_realview_type;
    typedef const_Matrix<Tp, unspecified> imagview_type;
    typedef imagview_type const_imagview_type;
```

```
// constructors, copy, and destructor
// [view.matrix.constructors]
const_Matrix(length_type num_of_rows, length_type num_of_columns,
             T const& value);
const_Matrix(length_type num_of_rows, length_type num_of_columns);
const_Matrix(Block&) VSIP_NOTHROW;
const_Matrix(const_Matrix const&) VSIP_NOTHROW;
const_Matrix(Matrix const&) VSIP_NOTHROW;
~const_Matrix() VSIP_NOTHROW;

// value accessors
// [view.matrix.valaccess]
value_type get(index_type, index_type) const VSIP_NOTHROW;
const_reference_type operator()(index_type, index_type) const
    VSIP_NOTHROW;

// subview accessors
// [view.matrix.subviews]
const_subview_type
    get(Domain<2> const&) const
        VSIP_THROW((std::bad_alloc));
const_subview_type
    operator()(Domain<2> const&) const
        VSIP_THROW((std::bad_alloc));
const_col_type col(index_type column_index) const
    VSIP_THROW((std::bad_alloc));
const_diag_type diag(index_difference_type) const
    VSIP_THROW((std::bad_alloc));
const_row_type row(index_type row_index) const
    VSIP_THROW((std::bad_alloc));
const_transpose_type transpose() const
    VSIP_THROW((std::bad_alloc));
// present only for complex type T:
const_realview_type
    real() const VSIP_THROW((std::bad_alloc));
const_imagview_type
    imag() const VSIP_THROW((std::bad_alloc));

// accessors
// [view.matrix.accessors]
block_type const& block() const VSIP_NOTHROW;
length_type size() const VSIP_NOTHROW;
length_type size(dimension_type) const VSIP_NOTHROW;
};
```

```

template <typename T = VSIP_DEFAULT_VALUE_TYPE,
        class Block = Dense<2, T> >
class Matrix {
public:
    // compile-time values
    static dimension_type const dim = 2;
    typedef Block block_type;
    typedef typename block_type::value_type value_type;
    typedef typename block_type::reference_type reference_type;
    typedef typename block_type::const_reference_type
        const_reference_type;

    // subview types
    // [view.matrix.subview_types]
    typedef Matrix<T, unspecified> subview_type;
    typedef const_Matrix<T, unspecified> const_subview_type;
    typedef Vector<T, unspecified> col_type;
    typedef const_Vector<T, unspecified> const_col_type;
    typedef Vector<T, unspecified> diag_type;
    typedef const_Vector<T, unspecified> const_diag_type;
    typedef Vector<T, unspecified> row_type;
    typedef const_Vector<T, unspecified> const_row_type;
    typedef Matrix<T, unspecified> transpose_type;
    typedef const_Matrix<T, unspecified> const_transpose_type;
    // present only for complex type T = complex<Tp>:
    typedef Matrix<Tp, unspecified> realview_type;
    typedef const_Matrix<Tp, unspecified> const_realview_type;
    typedef Matrix<Tp, unspecified> imagview_type;
    typedef const_Matrix<Tp, unspecified> const_imagview_type;

    // constructors, copy, assignment, and destructor
    // [view.matrix.constructors]
    Matrix(length_type num_of_rows, length_type num_of_columns,
           T const& value);
    Matrix(length_type num_of_rows, length_type num_of_columns);
    Matrix(Block&) VSIP_NOTHROW;
    Matrix(Matrix const&) VSIP_NOTHROW;
    template <typename T0, typename Block0>
    Matrix(Matrix<T0, Block0> const&);
    template <typename T0, typename Block0>
    Matrix& operator=(const_Matrix<T0, Block0>) VSIP_NOTHROW;
    Matrix& operator=(const_reference_type) VSIP_NOTHROW;
    template <typename T0>
    Matrix& operator=(T0 const&) VSIP_NOTHROW;

```

```
~Matrix() VSIP_NOTHROW;

// assignment operators
// [view.matrix.assign]
template <typename T0>
Matrix& operator+=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Matrix& operator+=(const_Matrix<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Matrix& operator+=(Matrix<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Matrix& operator==(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Matrix& operator==(const_Matrix<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Matrix& operator==(Matrix<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Matrix& operator*=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Matrix& operator*=(const_Matrix<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Matrix& operator*=(Matrix<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Matrix& operator/=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Matrix& operator/=(const_Matrix<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Matrix& operator/=(Matrix<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Matrix& operator&=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Matrix& operator&=(const_Matrix<T0, Block0>)
    VSIP_NOTHROW;
```

```

|   template <typename T0, typename Block0>
|   Matrix& operator+=(Matrix<T0, Block0>)
|       VSIP_NOTHROW;

|   template <typename T0>
|   Matrix& operator|=(T0 const&) VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Matrix& operator|=(const_Matrix<T0, Block0>)
|       VSIP_NOTHROW;

|   template <typename T0, typename Block0>
|   Matrix& operator|=(Matrix<T0, Block0>)
|       VSIP_NOTHROW;

|   template <typename T0>
|   Matrix& operator^=(T0 const&) VSIP_NOTHROW;
|   template <typename T0, typename Block0>
|   Matrix& operator^=(const_Matrix<T0, Block0>)
|       VSIP_NOTHROW;

|   template <typename T0, typename Block0>
|   Matrix& operator^=(Matrix<T0, Block0>)
|       VSIP_NOTHROW;

|
|   // value accessors
|   // [view.matrix.valaccess]
|   value_type get(index_type, index_type) const VSIP_NOTHROW;
|   unspecified put(index_type, index_type, T const&) VSIP_NOTHROW;
|   reference_type operator()(index_type, index_type) VSIP_NOTHROW;
|   const_reference_type operator()(index_type, index_type) const
|       VSIP_NOTHROW;

|
|   // subview accessors
|   // [view.matrix.subviews]
|   const_subview_type
|       get(Domain<2> const&) const
|       VSIP_THROW((std::bad_alloc));
|   subview_type
|       operator()(Domain<2> const&)
|       VSIP_THROW((std::bad_alloc));
|   const_subview_type
|       operator()(Domain<2> const&) const
|       VSIP_THROW((std::bad_alloc));
|   col_type col(index_type column_index)
|       VSIP_THROW((std::bad_alloc));
|   const_col_type col(index_type column_index) const
|       VSIP_THROW((std::bad_alloc));

```

```
diag_type diag(index_difference_type = 0)
    VSIP_THROW((std::bad_alloc));
const_diag_type diag(index_difference_type) const
    VSIP_THROW((std::bad_alloc));
row_type row(index_type row_index)
    VSIP_THROW((std::bad_alloc));
const_row_type row(index_type row_index) const
    VSIP_THROW((std::bad_alloc));
transpose_type transpose()
    VSIP_THROW((std::bad_alloc));
const_transpose_type transpose() const
    VSIP_THROW((std::bad_alloc));
// present only for complex type T:
realview_type real() VSIP_THROW((std::bad_alloc));
const_realview_type
    real() const VSIP_THROW((std::bad_alloc));
imagview_type imag() VSIP_THROW((std::bad_alloc));
const_imagview_type
    imag() const VSIP_THROW((std::bad_alloc));

// accessors
// [view.matrix.accessors]
block_type& block() const VSIP_NOTHROW;
length_type size() const VSIP_NOTHROW;
length_type size(dimension_type) const VSIP_NOTHROW;
};

// a specialization
template <typename Dim0 = Block, ...,
          typename Dimn = Block>
    // exactly VSIP_MAX_DIMENSION template parameters
class Matrix<T, Dense<2, T, Map<Dim0, ..., Dimn> > > {
public:

    // ... All members are the same as for Matrix<T,
    // Block> except Matrix's first two constructors are
    // replaced by:

    Matrix(length_type, length_type, T const& value,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Matrix(length_type, length_type,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Matrix(length_type, length_type, T const& value,
```

```

        const_Vector<processor_type>,
        Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Matrix(length_type, length_type,
        const_Vector<processor_type>,
        Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Matrix(length_type, length_type, T const& value, processor_type const*,
        Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
Matrix(length_type, length_type, processor_type const*,
        Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
    VSIP_THROW((std::bad_alloc));
};

// view conversion
template <template <typename, typename> class View,
        typename T, typename Block>
class ViewConversion;
}

```

6.5.1 *Template parameters***[view.matrix.template]**

1 `T` specifies the type of values stored in the *Matrix* object which has an associated block with type `Block` for storing the values. The only specializations which must be supported have `T` the same as `scalar_f`, `scalar_i`, `cscalar_f`, `cscalar_i`, or `bool`. An implementation is permitted to prevent instantiation for other choices of `T`.

2 `Block`

Requires: `T` must be `Block::value_type`. `Block` must be a two-dimensional block.

Note: `Block` need not be modifiable.

6.5.2 *Subview Types***[view.matrix.subview_types]**

1 `SubviewType` specifies the type of a two-dimensional subview of a *Matrix*. The type is a *Matrix* with value type `T` and an unspecified block type.

- 2 `const_subview_type` specifies the type of a non-modifiable two-dimensional subview of a *Matrix*. The type is a `const_Matrix` with value type `T` and an unspecified block type.
- 3 `col_type` specifies the type of a column subview of a *Matrix*. The type is a *Vector* with value type `T` and an unspecified block type.
- 4 `const_col_type` specifies the type of a non-modifiable column subview of a *Matrix*. The type is a `const_Vector` with value type `T` and an unspecified block type.
- 5 `diag_type` specifies the type of a diagonal subview of a *Matrix*. The type is a *Vector* with value type `T` and an unspecified block type.
- 6 `const_diag_type` specifies the type of a non-modifiable diagonal subview of a *Matrix*. The type is a `const_Vector` with value type `T` and an unspecified block type.
- 7 `row_type` specifies the type of a row subview of a *Matrix*. The type is a *Vector* with value type `T` and an unspecified block type.
- 8 `const_row_type` specifies the type of a non-modifiable row subview of a *Matrix*. The type is a `const_Vector` with value type `T` and an unspecified block type.
- 9 `transpose_type` specifies the type of a two-dimensional transpose subview of a *Matrix*. The type is a *Matrix* with value type `T` and an unspecified block type.
- 10 `const_transpose_type` specifies the type of a non-modifiable two-dimensional transpose subview of a *Matrix*. The type is a `const_Matrix` with value type `T` and an unspecified block type.
- 11 `realview_type` specifies the type of a subview of a *Matrix* containing only the real parts of the *Matrix*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Matrix* with value type `Tp` and an unspecified block type.
- 12 `const_realview_type` specifies the type of a non-modifiable subview of a *Matrix* containing only the real parts of the *Matrix*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Matrix` with value type `Tp` and an unspecified block type.

- 13 `imagview_type` specifies the type of a subview of a *Matrix* containing only the imaginary parts of the *Matrix*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Matrix* with value type `Tp` and an unspecified block type.
- 14 `const_imagview_type` specifies the type of a non-modifiable subview of a *Matrix* containing only the imaginary parts of the *Matrix*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Matrix` with value type `Tp` and an unspecified block type.

6.5.3 Constructors, copy, assignment, and destructor [view.matrix.constructors]

```
Matrix(
    length_type number_of_rows,
    length_type number_of_columns,
    T const& value);
```

Requires: `row_length > 0` and `column_length > 0`. Block must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Matrix(Block(Domain<2>(number_of_rows, number_of_columns), value, Block::map_type()))`.

Notes: Blocks are created with the effect of `increment_count`, `Matrix` does not invoke `increment_count` again for blocks it allocates.

```
Matrix(
    length_type number_of_rows,
    length_type number_of_columns);
```

Requires: `row_length > 0` and `column_length > 0`. Block must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Matrix(Block(Domain<2>(number_of_rows, number_of_columns), Block::map_type()))`.

Notes: Blocks are created with the effect of `increment_count`, `Matrix` does not invoke `increment_count` again for blocks it allocates.

```
Matrix(Block& block) VSIP_NOTHROW;
```

Requires: 2-dimensional modifiable `block`.

Effects: Constructs a `Matrix m` with associated block `block`. `m.size(0) == block.size(2,0)`. `m.size(1) == block.size(2,1)`.

```
const_Matrix(Block& block) VSIP_NOTHROW;
```

Requires: 2-dimensional block .

Effects: Constructs a `const_Matrix` `m` with associated block `block` .
`m.size(0) == block.size(2,0)` . `m.size(1) == block.size(2,1)` .

```
Matrix(Matrix const& m) VSIP_NOTHROW;
```

Effects: Constructs a subview *Matrix* object of `m` such that its domain is the same as `m`'s domain.

```
const_Matrix(Matrix const& m) VSIP_NOTHROW;
```

Effects: Constructs a subview `const_Matrix` object of `m` such that its domain is the same as `m`'s domain.

```
template <typename T0,  
         typename Block0>  
Matrix(Matrix<T0, Block0> const& m);
```

Requires: Block must be allocatable. The only specializations which must be supported are for `T0` the same as `T` . An implementation is permitted to prevent instantiation for other choices of `T0` . Type `T0` must be assignable to `T` .

Effects: Identical to `Matrix(Block(Domain<2>(m.size(0), m.size(1)), Block::map_type()))`; `*this = m`;

```
template <typename T0,  
         typename Block0>  
Matrix<T,Block0>&  
operator=(const_Matrix<T0,Block0> m) VSIP_NOTHROW;
```

Requires: `m` must be element-conformant with `*this` . `*this` and `m` must be order-independent assignment operands. The only specializations which must be supported are for `T0` the same as `T` . An implementation is permitted to prevent instantiation for other choices of `T0` . Type `T0` must be assignable to `T` .

Returns: `*this` .

Postconditions: For all `Index<2>` positions $\{(idx0, idx1)\}$ in the domain, `m.get(idx0, idx1) == this->get(idx0, idx1)`.

```
Matrix<T,Block>&
operator=(const_reference_type val)
VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation if `T` is not `scalar_f`, `scalar_i`, or `cscalar_f`.

Returns: `*this`.

Postconditions: For all `Index<2>` positions $\{(idx0, idx1)\}$ in the domain, `this->get(idx0, idx1) == val`.

Note: This function corresponds to VSIPL functions `vsip_mfill_i`, `vsip_mfill_f`, and `vsip_cmfill_f`.

```
template <typename T0>
Matrix<T,Block>&
operator=(T0 const& val) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation for `T0` different than `T`. Type `T0` must be assignable to `T`.

Returns: `*this`.

Postconditions: For all `Index<2>` positions $\{(idx0, idx1)\}$ in the domain, `this->get(idx0, idx1) == val`.

```
~ Matrix() VSIP_NOTHROW;
```

Effects: If this object is the only one using its block, the block is deallocated. Otherwise, its block's use count is decremented by one. Regardless, `*this` is deallocated.

6.5.4 Assignment operators

[view.matrix.assign]

```
template <typename T0>
Matrix&
operator+=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T the same as `scalar_f` or `scalar_i`, the only specializations which must be supported are for $T0$ the same as T . For T the same as `cscalar_f`, the only specializations which must be supported are for $T0$ the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations. $T0$ must be assignable to T .

Effects: For all index positions (`idx0`, `idx1`) in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) + v)`.

Returns: `*this`.

Note: This function corresponds to some of the functionality of VSIPL functions `vsip_smadd_i`, `vsip_smadd_f`, `vsip_rscmadd_f`, and `vsip_csmadd_f`.

```
template <typename T0,  
         typename Block0>  
Matrix&  
operator+=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Matrix&  
operator+=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: `m` must be element-conformant with `*this`. `*this` and `m` must be order-independent assignment operands. The only specializations which must be supported are for $T0$ the same as T . An implementation is permitted to prevent instantiation for other choices of $T0$. An implementation is permitted to prevent instantiation for T not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. $T0$ must be assignable to T .

Effects: For all index positions (`idx0`, `idx1`) in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) + m.get(idx0, idx1))`.

Returns: `*this`.

```
template <typename T0>  
Matrix&  
operator-=(T0 const& v) VSIP_NOTHROW;
```

Requires: For T the same as `scalar_f` or `scalar_i`, the only specializations which must be supported are for $T0$ the same as T . For T the same as

`cscalar_f`, the only specializations which must be supported are for `T0` the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`) in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) - v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
Matrix&
operator--(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Matrix&
operator--(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: `m` must be element-conformant with `*this`. `*this` and `m` must be order-independent assignment operands. The only specializations which must be supported are for `T0` the same as `T`. An implementation is permitted to prevent instantiation for other choices of `T0`. An implementation is permitted to prevent instantiation for `T` not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`) in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) - m.get(idx0, idx1))`.

Returns: `*this`.

```
template <typename T0>
Matrix&
operator*=(T0 const& v) VSIP_NOTHROW;
```

Requires: For `T` equal to `scalar_f`, the only specialization which must be supported is for `T0` the same as `T`. For `T` the same as `cscalar_f`, the only specializations which must be supported are for `T0` the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`) in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) * v)`.

Returns: `*this`.

```
template <typename T0,  
         typename Block0>  
Matrix&  
operator*=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Matrix&  
operator*=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: `m` must be element-conformant with `*this`. `*this` and `m` must be order-independent assignment operands. The only specializations which must be supported are for `T0` the same as `T`. An implementation is permitted to prevent instantiation for other choices of `T0`. An implementation is permitted to prevent instantiation for `T` not the same as `scalar_i`, `scalar_f`, or `cscalar_f`. `T0` must be assignable to `T`.

Effects: For all index positions `(idx0, idx1)` in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) * m.get(idx0, idx1))`.

Returns: `*this`.

```
template <typename T0>  
Matrix&  
operator/=(T0 const& v) VSIP_NOTHROW;
```

Requires: For `T` equal to `scalar_f`, the only specialization which must be supported is for `T0` the same as `T`. For `T` the same as `cscalar_f`, the only specialization which must be supported is for `T0` the same as `scalar_f`. An implementation is permitted to prevent other instantiations. `T0` must be assignable to `T`.

Effects: For all index positions `(idx0, idx1)` in `*this`'s domain, `this->put(idx0, idx1, this->get(idx0, idx1) / v)`.

Returns: `*this`.

```
template <typename T0,  
         typename Block0>  
Matrix&
```

```
operator/=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
```

```
Matrix&
```

```
operator/=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: *m* must be element-conformant with **this*. **this* and *m* must be order-independent assignment operands. The only specializations which must be supported are for *T0* the same as *T*. An implementation is permitted to prevent instantiation for other choices of *T0*. An implementation is permitted to prevent instantiation for *T* not the same as *scalar_i*, *scalar_f*, or *cscalar_f*. *T0* must be assignable to *T*.

Effects: For all index positions (*idx0*, *idx1*) in **this*'s domain, *this->put*(*idx0*, *idx1*, *this->get*(*idx0*, *idx1*) / *m.get*(*idx0*, *idx1*)).

Returns: **this*.

```
template <typename T0>
```

```
Matrix&
```

```
operator&=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. *T0* must be assignable to *T*.

Effects: For all index positions (*idx0*, *idx1*) in **this*'s domain, *this->put*(*idx0*, *idx1*, *this->get*(*idx0*, *idx1*) & *v*).

Returns: **this*.

```
template <typename T0,
          typename Block0>
```

```
Matrix&
```

```
operator&=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
```

```
Matrix&
```

```
operator&=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: *m* must be element-conformant with **this*. **this* and *m* must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of *T0*. *T0* must be assignable to *T*.

Effects: For all index positions (*idx0*, *idx1*) in **this*'s domain, *this->put(idx0, idx1, this->get(idx0, idx1) & m.get(idx0, idx1))*.

Returns: **this*.

```
template <typename T0>
Matrix&
operator|=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. *T0* must be assignable to *T*.

Effects: For all index positions (*idx0*, *idx1*) in **this*'s domain, *this->put(idx0, idx1, this->get(idx0, idx1) | v)*.

Returns: **this*.

```
template <typename T0,
          typename Block0>
Matrix&
operator|=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Matrix&
operator|=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: *m* must be element-conformant with **this*. **this* and *m* must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of *T0*. *T0* must be assignable to *T*.

Effects: For all index positions (*idx0*, *idx1*) in **this*'s domain, *this->put(idx0, idx1, this->get(idx0, idx1) | m.get(idx0, idx1))*.

Returns: **this*.

```
template <typename T0>
Matrix&
operator^=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. *T0* must be assignable to *T*.

Effects: For all index positions (idx0, idx1) in *this's domain, this->put(idx0, idx1, this->get(idx0, idx1) ^ v).

Returns: *this.

```
template <typename T0,
          typename Block0>
Matrix&
operator^=(const_Matrix<T0, Block0> m) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Matrix&
operator^=(Matrix<T0, Block0> m) VSIP_NOTHROW;
```

Requires: m must be element-conformant with *this. *this and m must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of T0. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1) in *this's domain, this->put(idx0, idx1, this->get(idx0, idx1) ^ m.get(idx0, idx1)).

Returns: *this.

6.5.5 Value accessors

[**view.matrix.valaccess**]

[*Note:* The restrictions for get and put follow from the view requirements in [view.view]. Only additional restrictions occur here.]

```
reference_type
operator () (index_type idx0, index_type idx1) VSIP_NOTHROW;
```

Requires: This value accessor is provided only by Matrix, not const_Matrix. idx0 < this->size(0) and idx1 < this->size(1). An implementation should provide this accessor but is not required to do so.

Returns: Value at the Index<2> position (idx0,idx1).

Note: A VSIPL++ Library implemented using a VSIPL implementation cannot provide this accessor because VSIPL does not provide such functionality.

6.5.6. [VIEW.MATRIX.SUBVIEWS] SUBVIEWS

```
const_reference_type  
operator () (index_type idx0, index_type idx1)  
const VSIP_NOTHROW;
```

Requires: $\text{idx0} < \text{this->\text{size}(0)}$ and $\text{idx1} < \text{this->\text{size}(1)}$. An implementation should provide this accessor but is not required to do so.

Returns: Value at the `Index<2>` position $(\text{idx0}, \text{idx1})$.

Note: A VSIP++ Library implemented using a VSIP implementation can provide this accessor but not its non-const version.

6.5.6 Subviews

[view.matrix.subviews]

- 1 These functions return subviews of a `const_Matrix` or `Matrix` object.

```
const_subview_type  
get (Domain<2> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: d is a subset of the *Matrix*'s domain.

Returns: A subview *Matrix* object of **this* with domain `Domain<2>(d[0].size(), d[1].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
subview_type  
operator () (Domain<2> const& d) VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a *Matrix* object. d is a subset of the *Matrix*'s domain.

Returns: A subview *Matrix* object of **this* with domain `Domain<2>(d[0].size(), d[1].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_subview_type  
operator () (Domain<2> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: d is a subset of the *Matrix*'s domain.

Returns: A subview *Matrix* object of **this* with domain `Domain<2>(d[0].size(), d[1].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
col_type
col(index_type column_index) VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Matrix` object. `column_index < this->size(1)`.

Returns: A subview of the `Matrix` object containing the specified column number `column_index` with domain `Domain<1>(this->size(0))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_col_type
col(index_type column_index) const VSIP_THROW((std::bad_alloc));
```

Requires: `column_index < this->size(1)`.

Returns: A subview of the `Matrix` object containing the specified column number `column_index` with domain `Domain<1>(this->size(0))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
diag_type
diag(index_difference_type diagonal_offset = 0)
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Matrix` object. If `diagonal_offset` is positive, `diagonal_offset < this->size(1)`. If `diagonal_offset` is negative, `-diagonal_offset < this->size(0)`.

Returns: If `diagonal_offset == 0`, a subview of the `Matrix` object containing its diagonal values. If `diagonal_offset > 0`, return a subview of the `Matrix` object containing values in the diagonal `diagonal_offset` above the main diagonal. Its domain is `Domain<1>(min(this->size(0), this->size(1) - diagonal_offset))`. If `diagonal_offset < 0`, return a subview of the `Matrix` object containing values in the diagonal `diagonal_offset` below the main diagonal. Its domain is `Domain<1>(min(this->size(0) + diagonal_offset, this->size(1)))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_diag_type  
diag(index_difference_type diagonal_offset = 0)  
const VSIP_THROW((std::bad_alloc));
```

Requires: If `diagonal_offset` is positive, `diagonal_offset < this->size(1)`. If `diagonal_offset` is negative, `-diagonal_offset < this->size(0)`.

Returns: If `diagonal_offset == 0`, a subview of the *Matrix* object containing its diagonal values. If `diagonal_offset > 0`, return a subview of the *Matrix* object containing values in the diagonal `diagonal_offset` above the main diagonal. Its domain is `Domain<1>(min(this->size(0), this->size(1) - diagonal_offset))`. If `diagonal_offset < 0`, return a subview of the *Matrix* object containing values in the diagonal `diagonal_offset` below the main diagonal. Its domain is `Domain<1>(min(this->size(0) + diagonal_offset, this->size(1)))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
row_type  
row(index_type row_index)  
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a *Matrix* object. `row_index < this->size(0)`.

Returns: A subview of the *Matrix* object containing the specified row number `row_index` and having domain `Domain<1>(this->size(1))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_row_type  
row(index_type row_index) const  
VSIP_THROW((std::bad_alloc));
```

Requires: `row_index < this->size(0)`.

Returns: A subview of the *Matrix* object containing the specified row number `row_index` and having domain `Domain<1>(this->size(1))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
transpose_type
transpose()
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a *Matrix* object.

Returns: A subview of the *Matrix* object containing its transpose and having domain `Domain<2>(this->size(1), this->size(0))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

Note: The result is a subview of the original view so modifying values in the original view also changes values in the subview. Many implementations will choose to implement this subview by reordering indices. See `[math.matvec.transpose]` for an alternate transpose implementation.

```
const_transpose_type
transpose() const
VSIP_THROW((std::bad_alloc));
```

Returns: A subview of the *Matrix* object containing its transpose and having domain `Domain<2>(this->size(1), this->size(0))`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

Note: The result is a subview of the original view. Many implementations will choose to implement this subview by reordering indices. See `[math.matvec.transpose]` for an alternate transpose implementation.

```
const_realview_type
real() const VSIP_THROW((std::bad_alloc));
```

Requires: `T` must be a complex type.

Returns: A subview `const_Matrix` object of `*this` with the same domain but accessing only the real parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
realview_type  
real() VSIP_THROW((std::bad_alloc));
```

Requires: T must be a complex type.

Returns: A subview *Matrix* object of *this with the same domain but accessing only the real parts of the complex values of the object.

Throws: std::bad_alloc indicating memory allocation for the underlying block failed.

```
const_imagview_type  
imag() const VSIP_THROW((std::bad_alloc));
```

Requires: T must be a complex type.

Returns: A subview const_Matrix object of *this with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: std::bad_alloc indicating memory allocation for the underlying block failed.

```
imagview_type  
imag() VSIP_THROW((std::bad_alloc));
```

Requires: T must be a complex type.

Returns: A subview *Matrix* object of *this with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: std::bad_alloc indicating memory allocation for the underlying block failed.

6.5.7 Accessors

[view.matrix.accessors]

```
block_type const&  
block() const VSIP_NOTHROW;  
block_type&  
block() const VSIP_NOTHROW;
```

Returns: The matrix's underlying block. const_Matrix::block() returns a block_type const& while Matrix::block() returns a block_type &.

```
length_type  
size() const VSIP_NOTHROW;
```

Returns: The number of values in the matrix.

```
length_type
size(dimension_type d) const VSIP_NOTHROW;
```

Requires: $0 \leq d \leq 1$.

Returns: The number of values in the specified dimension.

6.5.8 Matrix type conversion

[[view.matrix.convert](#)]

- 1 [Note: The class `ViewConversion` converts between constant and non-constant view classes.]
- 2 `ViewConversion<Matrix, T, Block>::const_view_type` equals `const_Matrix<T, Block>` and `ViewConversion<Matrix, T, Block>::view_type` equals `Matrix<T, Block>`.

6.6 Tensor

[[view.tensor](#)]

- 1 The `const_Tensor` and `Tensor` classes are views implementing the mathematical idea of tensors. A `const_Tensor` view is not modifiable, but a `Tensor` view is modifiable.
- 2 The interfaces for `const_Tensor` and `Tensor` are similar so they are simultaneously specified except where noted. For these, *Tensor* indicates both `const_Tensor` and `Tensor`. The term “tensor” refers to a `const_Tensor` or `Tensor` object.

Header <vsip/tensor.hpp> synopsis

```
namespace vsip {
  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<3, T> >
    class const_Tensor;

  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<3, T> >
    class Tensor;

  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           class Block = Dense<3, T> >
```

```
class const_Tensor {
public:
    // compile-time values
    static dimension_type const dim = 3;
    typedef Block block_type;
    typedef typename block_type::value_type value_type;
    typedef typename block_type::reference_type reference_type;
    typedef typename block_type::const_reference_type
        const_reference_type;

    // subview types
    // [view.tensor.subview_types]
    typedef const_Tensor<T, unspecified> subview_type;
    typedef subview_type const_subview_type;
|   template <dimension_type D1, dimension_type D2>
|   struct subvector
|   {
|       typedef const_Vector<T, unspecified> type;
|       typedef type const_type;
|       typedef const_Vector<T, unspecified> subset_type;
|       typedef subset_type const_subset_type;
|   };
|   template <dimension_type D>
|   struct submatrix
|   {
|       typedef const_Matrix<T, unspecified> type;
|       typedef type const_type;
|       typedef const_Matrix<T, unspecified> subset_type;
|       typedef subset_type const_subset_type;
|   };
|   template <dimension_type D0, dimension_type D1, dimension_type D2>
|   struct transpose_view
|   {
|       typedef const_Tensor<T, unspecified> type;
|       typedef type const_type;
|   };
|   // present only for complex type T = complex<Tp>:
    typedef const_Tensor<Tp, unspecified> realview_type;
    typedef realview_type const_realview_type;
    typedef const_Tensor<Tp, unspecified> imagview_type;
    typedef imagview_type const_imagview_type;

    // constructors, copy, assignment, and destructor
    // [view.tensor.constructors]
    const_Tensor(length_type z_length, length_type y_length,
```

```

        length_type x_length, T const& value);
const_Tensor(length_type z_length, length_type y_length,
             length_type x_length);
const_Tensor(Block&) VSIP_NOTHROW;
const_Tensor(const_Tensor const&) VSIP_NOTHROW;
const_Tensor(Tensor const&) VSIP_NOTHROW;
~const_Tensor() VSIP_NOTHROW;

// transposition
// [view.tensor.transpose]
| template <dimension_type D0, dimension_type D1, dimension_type D2>
| typename transpose_view<D0, D1, D2>::const_type
| transpose() const VSIP_NOTHROW;

// value accessors
// [view.tensor.valaccess]
value_type get(index_type, index_type, index_type) const VSIP_NOTHROW;
const_reference_type operator()(index_type, index_type, index_type) const
    VSIP_NOTHROW;

// subview accessors
// [view.tensor.subviews]
const_subview_type
    get(Domain<3> const&) const VSIP_THROW((std::bad_alloc));
const_subview_type
    operator()(Domain<3> const&) const VSIP_THROW((std::bad_alloc));

| typename subvector<0, 1>::const_type
| operator()(index_type, index_type, whole_domain_type) const
| VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::const_type
| operator()(index_type, whole_domain_type, index_type) const
| VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::const_type
| operator()(whole_domain_type, index_type, index_type) const
| VSIP_THROW((std::bad_alloc));

| typename subvector<0, 1>::const_subset_type
| operator()(index_type, index_type, Domain<1> const&) const
| VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::const_subset_type
| operator()(index_type, Domain<1> const&, index_type) const
| VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::const_subset_type
| operator()(Domain<1> const&, index_type, index_type) const

```

```
    VSIP_THROW((std::bad_alloc));

|  typename submatrix<0>::const_type
|  operator()(index_type, whole_domain_type, whole_domain_type) const
|  VSIP_THROW((std::bad_alloc));
|  typename submatrix<1>::const_type
|  operator()(whole_domain_type, index_type, whole_domain_type) const
|  VSIP_THROW((std::bad_alloc));
|  typename submatrix<2>::const_type
|  operator()(whole_domain_type, whole_domain_type, index_type) const
|  VSIP_THROW((std::bad_alloc));

|  typename submatrix<0>::const_subset_type
|  operator()(index_type, Domain<1> const&, Domain<1> const&) const
|  VSIP_THROW((std::bad_alloc));
|  typename submatrix<1>::const_subset_type
|  operator()(Domain<1> const&, index_type, Domain<1> const&) const
|  VSIP_THROW((std::bad_alloc));
|  typename submatrix<2>::const_subset_type
|  operator()(Domain<1> const&, Domain<1> const&, index_type) const
|  VSIP_THROW((std::bad_alloc));

// present only for complex type T:
const_realview_type
    real() const VSIP_THROW((std::bad_alloc));
const_imagview_type
    imag() const VSIP_THROW((std::bad_alloc));

// accessors
// [view.tensor.accessors]
block_type& block() const VSIP_NOTHROW;
length_type size() const VSIP_NOTHROW;
length_type size(dimension_type) const VSIP_NOTHROW;
};

template <typename T = VSIP_DEFAULT_VALUE_TYPE,
          class Block = Dense<3, T> >
class Tensor {
public:
    // compile-time values
    static dimension_type const dim = 3;
    typedef Block block_type;
    typedef typename block_type::value_type value_type;
    typedef typename block_type::reference_type reference_type;
    typedef typename block_type::const_reference_type
```

```

        const_reference_type;

// subview types
// [view.tensor.subview_types]
typedef Tensor<T, unspecified> subview_type;
typedef const_Tensor<T, unspecified> const_subview_type;
| template <dimension_type D1, dimension_type D2>
| struct subvector
| {
|     typedef Vector<T, unspecified>         type;
|     typedef const_Vector<T, unspecified>   const_type;
|     typedef Vector<T, unspecified>         subset_type;
|     typedef const_Vector<T, unspecified>   const_subset_type;
| };
| template <dimension_type D>
| struct submatrix
| {
|     typedef Matrix<T, unspecified>         type;
|     typedef const_Matrix<T, unspecified>   const_type;
|     typedef Matrix<T, unspecified>         subset_type;
|     typedef const_Matrix<T, unspecified>   const_subset_type;
| };
| template <dimension_type D0, dimension_type D1, dimension_type D2>
| struct transpose_view
| {
|     typedef Tensor<T, unspecified>         type;
|     typedef const_Tensor<T, unspecified>   const_type;
| };
| // present only for complex type T = complex<Tp>:
| typedef Tensor<Tp, unspecified> realview_type;
| typedef const_Tensor<Tp, unspecified> const_realview_type;
| typedef Tensor<Tp, unspecified> imagview_type;
| typedef const_Tensor<Tp, unspecified> const_imagview_type;

// constructors, copy, assignment, and destructor
// [view.tensor.constructors]
Tensor(length_type z_length, length_type y_length,
        length_type x_length, T const& value);
Tensor(length_type z_length, length_type y_length,
        length_type x_length);
Tensor(Block&) VSIP_NOTHROW;
Tensor(Tensor const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Tensor(Tensor<T0, Block0> const&);
template <typename T0, typename Block0>

```

```
Tensor& operator=(const_Tensor<T0,Block0>) VSIP_NOTHROW;
Tensor& operator=(const_reference_type) VSIP_NOTHROW;
template <typename T0>
Tensor& operator=(T0 const&) VSIP_NOTHROW;
~Tensor() VSIP_NOTHROW;

// assignment operators
// [view.tensor.assign]
template <typename T0>
Tensor& operator+=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Tensor& operator+=(const_Tensor<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Tensor& operator+=(Tensor<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Tensor& operator-=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Tensor& operator-=(const_Tensor<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Tensor& operator-=(Tensor<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Tensor& operator*=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Tensor& operator*=(const_Tensor<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Tensor& operator*=(Tensor<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
Tensor& operator/=(T0 const&) VSIP_NOTHROW;
template <typename T0, typename Block0>
Tensor& operator/=(const_Tensor<T0, Block0>)
    VSIP_NOTHROW;
| template <typename T0, typename Block0>
| Tensor& operator/=(Tensor<T0, Block0>)
|     VSIP_NOTHROW;

template <typename T0>
```

```

Tensor& operator+=(T0 const&) VSIP_NO_THROW;
template <typename T0, typename Block0>
Tensor& operator+=(const_Tensor<T0, Block0>)
    VSIP_NO_THROW;
|
template <typename T0, typename Block0>
|
Tensor& operator+=(Tensor<T0, Block0>)
|
    VSIP_NO_THROW;

template <typename T0>
Tensor& operator|=(T0 const&) VSIP_NO_THROW;
template <typename T0, typename Block0>
Tensor& operator|=(const_Tensor<T0, Block0>)
    VSIP_NO_THROW;
|
template <typename T0, typename Block0>
|
Tensor& operator|=(Tensor<T0, Block0>)
|
    VSIP_NO_THROW;

template <typename T0>
Tensor& operator^=(T0 const&) VSIP_NO_THROW;
template <typename T0, typename Block0>
Tensor& operator^=(const_Tensor<T0, Block0>)
    VSIP_NO_THROW;
|
template <typename T0, typename Block0>
|
Tensor& operator^=(Tensor<T0, Block0>)
|
    VSIP_NO_THROW;

// transposition
// [view.tensor.transpose]
|
template <dimension_type D0, dimension_type D1, dimension_type D2>
|
typename transpose_view<D0, D1, D2>::const_type
|
transpose() const VSIP_NO_THROW;
|
template <dimension_type D0, dimension_type D1, dimension_type D2>
|
typename transpose_view<D0, D1, D2>::type
|
transpose() VSIP_NO_THROW;

// value accessors
// [view.tensor.valaccess]
value_type get(index_type, index_type, index_type) const VSIP_NO_THROW;
unspecified put(index_type, index_type, index_type, T const&) VSIP_NO_THROW;
reference_type operator()(index_type, index_type, index_type) VSIP_NO_THROW;
const_reference_type operator()(index_type, index_type, index_type) const
    VSIP_NO_THROW;

// subview accessors
// [view.tensor.subviews]

```

```
const_subview_type
  get(Domain<3> const&) const VSIP_THROW((std::bad_alloc));
subview_type
  operator() (Domain<3> const&) VSIP_THROW((std::bad_alloc));
const_subview_type
  operator() (Domain<3> const&) const
  VSIP_THROW((std::bad_alloc));

| typename subvector<0, 1>::type
|   operator() (index_type, index_type, whole_domain_type)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 1>::const_type
|   operator() (index_type, index_type, whole_domain_type) const
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::type
|   operator() (index_type, whole_domain_type, index_type)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::const_type
|   operator() (index_type, whole_domain_type, index_type) const
|   VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::type
|   operator() (whole_domain_type, index_type, index_type)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::const_type
|   operator() (whole_domain_type, index_type, index_type) const
|   VSIP_THROW((std::bad_alloc));

| typename subvector<0, 1>::subset_type
|   operator() (index_type, index_type, Domain<1> const&)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 1>::const_subset_type
|   operator() (index_type, index_type, Domain<1> const&) const
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::subset_type
|   operator() (index_type, Domain<1> const&, index_type)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<0, 2>::const_subset_type
|   operator() (index_type, Domain<1> const&, index_type) const
|   VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::subset_type
|   operator() (Domain<1> const&, index_type, index_type)
|   VSIP_THROW((std::bad_alloc));
| typename subvector<1, 2>::const_subset_type
|   operator() (Domain<1> const&, index_type, index_type) const
|   VSIP_THROW((std::bad_alloc));
```

```

| typename submatrix<0>::type
|     operator()(index_type, whole_domain_type, whole_domain_type)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<0>::const_type
|     operator()(index_type, whole_domain_type, whole_domain_type) const
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<1>::type
|     operator()(whole_domain_type, index_type, whole_domain_type)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<1>::const_type
|     operator()(whole_domain_type, index_type, whole_domain_type) const
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<2>::type
|     operator()(whole_domain_type, whole_domain_type, index_type)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<2>::const_type
|     operator()(whole_domain_type, whole_domain_type, index_type) const
|     VSIP_THROW((std::bad_alloc));

|
| typename submatrix<0>::subset_type
|     operator()(index_type, Domain<1> const&, Domain<1> const&)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<0>::const_subset_type
|     operator()(index_type, Domain<1> const&, Domain<1> const&) const
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<1>::subset_type
|     operator()(Domain<1> const&, index_type, Domain<1> const&)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<1>::const_subset_type
|     operator()(Domain<1> const&, index_type, Domain<1> const&) const
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<2>::subset_type
|     operator()(Domain<1> const&, Domain<1> const&, index_type)
|     VSIP_THROW((std::bad_alloc));
| typename submatrix<2>::const_subset_type
|     operator()(Domain<1> const&, Domain<1> const&, index_type) const
|     VSIP_THROW((std::bad_alloc));

// present only for complex type T:
const_realview_type
    real() const VSIP_THROW((std::bad_alloc));
realview_type
    real() VSIP_THROW((std::bad_alloc));
const_imagview_type

```

```
    imag() const VSIP_THROW((std::bad_alloc));
    imagview_type
    imag() VSIP_THROW((std::bad_alloc));

    // accessors
    // [view.tensor.accessors]
    block_type& block() const VSIP_NOTHROW;
    length_type size() const VSIP_NOTHROW;
    length_type size(dimension_type) const VSIP_NOTHROW;
};

// a specialization
template <typename Dim0 = Block, ...,
          typename Dimn = Block>
    // exactly VSIP_MAX_DIMENSION template parameters
    class Tensor<T, Dense<3, T, Map<Dim0, ..., Dimn> > > {
public:

    // ... All members are the same as for Tensor<T,
    // Block> except Tensor's first two constructors are
    // replaced by:

    Tensor(length_type, length_type, length_type, T const& value,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Tensor(length_type, length_type, length_type,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Tensor(length_type, length_type, length_type,
           T const& value, const_Vector<processor_type>,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Tensor(length_type, length_type, length_type,
           const_Vector<processor_type>,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Tensor(length_type, length_type, length_type,
           T const& value, processor_type const*,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
    Tensor(length_type, length_type, length_type,
           processor_type const*,
           Dim0 const& = Dim0(), ..., Dimn const& = Dimn())
        VSIP_THROW((std::bad_alloc));
};
```

```

// view conversion
template <template <typename, typename> class View,
    typename T, typename Block>
class ViewConversion;
}

```

6.6.1 Template parameters

[view.tensor.template]

- 1 T specifies the type of values stored in the *Tensor* object which has an associated block with type *Block* for storing the values. The only specializations which must be supported have T the same as *scalar_f*, *scalar_i*, *cscalar_f*, *cscalar_i*, or *bool*. An implementation is permitted to prevent instantiation for other choices of T.
- 2 *Block*

Requires: T must be *Block::value_type*. *Block* must be a three-dimensional block.

Note: *Block* need not be modifiable.

6.6.2 Subview Types

[view.tensor.subview_types]

- 1 *subview_type* specifies the type of a three-dimensional subview of a *Tensor*. The type is a *Tensor* with value type T and an unspecified block type.
- 2 *const_subview_type* specifies the type of a non-modifiable three-dimensional subview of a *Tensor*. The type is a *const_Tensor* with value type T and an unspecified block type.
- 3 *subvector<D1, D2>::type* (where D1 and D2 are *dimension_types*) specifies the type of a one-dimensional whole-domain subview of a *Tensor* with fixed-dimensions D1 and D2. The type is a *Vector* with value type T and an unspecified block type.
- 4 *subvector<D1, D2>::const_type* (where D1 and D2 are *dimension_types*) specifies the type of a non-modifiable one-dimensional whole-domain subview of a *Tensor* with fixed-dimensions D1 and D2. The type is a *const_Vector* with value type T and an unspecified block type.

- 5 `subvector<D1, D2>::subset_type` (where `D1` and `D2` are `dimension_types`) specifies the type of a one-dimensional subset-domain subview of a *Tensor* with fixed-dimensions `D1` and `D2`. The type is a *Vector* with value type `T` and an unspecified block type.
- 6 `subvector<D1, D2>::const_subset_type` (where `D1` and `D2` are `dimension_types`) specifies the type of a non-modifiable one-dimensional subset-domain subview of a *Tensor* with fixed-dimensions `D1` and `D2`. The type is a `const_Vector` with value type `T` and an unspecified block type.
- 7 `submatrix<D>::type` (where `D` is a `dimension_type`) specifies the type of a two-dimensional whole-domain subview of a *Tensor* with fixed-dimension `D`. The type is a *Matrix* with value type `T` and an unspecified block type.
- 8 `submatrix<D>::const_type` (where `D` is a `dimension_type`) specifies the type of a non-modifiable two-dimensional whole-domain subview of a *Tensor* with fixed-dimension `D`. The type is a `const_Matrix` with value type `T` and an unspecified block type.
- 9 `submatrix<D>::subset_type` (where `D` is a `dimension_type`) specifies the type of a two-dimensional subset-domain subview of a *Tensor* with fixed-dimension `D`. The type is a *Matrix* with value type `T` and an unspecified block type.
- 10 `submatrix<D>::const_subset_type` (where `D` is a `dimension_type`) specifies the type of a non-modifiable two-dimensional subset-domain subview of a *Tensor* with fixed-dimension `D`. The type is a `const_Matrix` with value type `T` and an unspecified block type.
- 11 `transpose_view<D0, D1, D2>::type` specifies the type of a three-dimensional transpose subview of a *Tensor* for dimension permutation `D0, D1, D2`. The type is a *Tensor* with value type `T` and an unspecified block type.
- 12 `transpose_view<D0, D1, D2>::const_type` specifies the type of a non-modifiable, three-dimensional transpose subview of a *Tensor* for dimension permutation `D0, D1, D2`. The type is a `const_Tensor` with value type `T` and an unspecified block type.
- 13 `realview_type` specifies the type of a subview of a *Tensor* containing only the real parts of the *Tensor*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Tensor* with value type `Tp` and an unspecified block type.

- 14 `const_realview_type` specifies the type of a non-modifiable subview of a *Tensor* containing only the real parts of the *Tensor*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Tensor` with value type `Tp` and an unspecified block type.
- 15 `imagview_type` specifies the type of a subview of a *Tensor* containing only the imaginary parts of the *Tensor*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a *Tensor* with value type `Tp` and an unspecified block type.
- 16 `const_imagview_type` specifies the type of a non-modifiable subview of a *Tensor* containing only the imaginary parts of the *Tensor*'s values. This type is defined only if `T` is a complex type `complex<Tp>`. The type is a `const_Tensor` with value type `Tp` and an unspecified block type.

6.6.3 Constructors, copy, assignment, and destructor [view.tensor.constructors]

```
Tensor (
    length_type z_length,
    length_type y_length,
    length_type x_length,
    T const& value);
```

Requires: `z_length > 0 && y_length > 0 && x_length > 0`.
Block must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Tensor(Block(Domain<3>(z_length, y_length, x_length), value, Block::map_type()))`.

Notes: Blocks are created with the effect of `increment_count`, `Tensor` does not invoke `increment_count` again for blocks it allocates.

```
Tensor (
    length_type z_length,
    length_type y_length,
    length_type x_length);
```

Requires: `z_length > 0 && y_length > 0 && x_length > 0`.
Block must be allocatable. `Block::map_type` must have a default constructor.

Effects: Identical to `Tensor(Block(Domain<3>(z_length, y_length, x_length), Block::map_type()))`.

Notes: Blocks are created with the effect of `increment_count`, `Tensor` does not invoke `increment_count` again for blocks it allocates.

```
Tensor(Block& block) VSIP_NOTHROW;
```

Requires: 3-dimensional modifiable `block`.

Effects: Constructs a `Tensor t` with associated `block`. `t.size(0) == block.size(3,0)`. `t.size(1) == block.size(3,1)`. `t.size(2) == block.size(3,2)`.

```
const_Tensor(Block& block) VSIP_NOTHROW;
```

Requires: 3-dimensional `block`.

Effects: Constructs a `const_Tensor t` with associated `block`. `t.size(0) == block.size(3,0)`. `t.size(1) == block.size(3,1)`. `t.size(2) == block.size(3,2)`.

```
Tensor(Tensor const& t) VSIP_NOTHROW;
```

Effects: Constructs a subview `Tensor` object of `t` such that its domain is the same as `t`'s domain.

```
const_Tensor(Tensor const& t) VSIP_NOTHROW;
```

Effects: Constructs a subview `const_Tensor` object of `t` such that its domain is the same as `t`'s domain.

```
template <typename T0,
          typename Block0>
Tensor(Tensor<T0, Block0> const& t);
```

Requires: `Block` must be allocatable. The only specifications which must be supported are for `T0` the same as `T`. An implementation is permitted to prevent instantiation. Type `T0` must be assignable to `T`.

Effects: Identical to `Tensor(Block(Domain<3>(t.size(0), t.size(1), t.size(2)), Block::map_type()))`; `*this = t`;

```

template <typename T0,
          typename Block0>
Tensor<T,Block>&
operator=(const_Tensor<T0,Block0> t) VSIP_NOTHROW;

```

Requires: `t` must be element-conformant with `*this`. `*this` and `t` must be order-independent assignment operands. An implementation is permitted to prevent instantiation. Type `T0` must be assignable to `T`.

Returns: `*this`.

Postconditions: For all `Index<3>` positions `(idx0, idx1, idx2)` in `*this`'s domain, `t.get(idx0,idx1,idx2) == this->get(idx0,idx1,idx2)`.

```

Tensor<T,Block>&
operator=(const_reference_type val)
VSIP_NOTHROW;

```

Requires: An implementation is permitted to prevent instantiation.

Returns: `*this`.

Postconditions: For all `Index<3>` positions `(idx0,idx1,idx2)` in `*this`'s domain, `this->get(idx0,idx1,idx2) == val`.

```

template <typename T0>
Tensor<T,Block>&
operator=(T0 const& val) VSIP_NOTHROW;

```

Requires: An implementation is permitted to prevent instantiation. Type `T0` must be assignable to `T`.

Returns: `*this`.

Postconditions: For all `Index<3>` positions `(idx0,idx1,idx2)` in `*this`'s domain, `this->get(idx0,idx1,idx2) == val`.

```

~ Tensor() VSIP_NOTHROW;

```

Effects: If this object is the only one using its block, the block is deallocated. Otherwise, its block's use count is decremented by one. Regardless, the object is deallocated.

1 Tensor, but not const_Tensor, has assignment operators.

```
template <typename T0>
    Tensor& operator+=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) + v).

Returns: *this.

```
template <typename T0,
          typename Block0>
Tensor&
operator+=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Tensor&
operator+=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: t must be element-conformant with *this. *this and t must be order-independent assignment operands. An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) + t.get(idx0, idx1, idx2)).

Returns: *this.

```
template <typename T0>
Tensor&
operator-=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) - v).

Returns: *this.

```
template <typename T0,
          typename Block0>
Tensor&
operator--(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Tensor&
operator--(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: t must be element-conformant with *this. *this and t must be order-independent assignment operands. An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) - t.get(idx0, idx1, idx2)).

Returns: *this.

```
template <typename T0>
Tensor&
operator*=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) * v).

Returns: *this.

```
template <typename T0,
          typename Block0>
Tensor&
operator*=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator*=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: t must be element-conformant with *this. *this and t must be order-independent assignment operands. An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) * t.get(idx0, idx1, idx2)).

Returns: *this.

```
template <typename T0>  
Tensor&  
operator/=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) / v).

Returns: *this.

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator/=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator/=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: t must be element-conformant with *this. *this and t must be order-independent assignment operands. An implementation is permitted to prevent instantiation. T0 must be assignable to T.

Effects: For all index positions (idx0, idx1, idx2) in *this's domain, this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) / t.get(idx0, idx1, idx2)).

Returns: `*this`.

```
template <typename T0>
Tensor&
operator&=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) & v)`.

Returns: `*this`.

```
template <typename T0,
          typename Block0>
Tensor&
operator&=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,
          typename Block0>
Tensor&
operator&=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: `t` must be element-conformant with `*this`. `*this` and `t` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) & t.get(idx0, idx1, idx2))`.

Returns: `*this`.

```
template <typename T0>
Tensor&
operator|=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) | v)`.

Returns: `*this`.

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator|=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator|=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: `t` must be element-conformant with `*this`. `*this` and `t` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) | t.get(idx0, idx1, idx2))`.

Returns: `*this`.

```
template <typename T0>  
Tensor&  
operator^=(T0 const& v) VSIP_NOTHROW;
```

Requires: An implementation is permitted to prevent instantiation. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) ^ v)`.

Returns: `*this`.

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator^=(const_Tensor<T0, Block0> t) VSIP_NOTHROW;
```

```
template <typename T0,  
         typename Block0>  
Tensor&  
operator^=(Tensor<T0, Block0> t) VSIP_NOTHROW;
```

Requires: `t` must be element-conformant with `*this`. `*this` and `t` must be order-independent assignment operands. An implementation is permitted to prevent instantiation for any choice of `T0`. `T0` must be assignable to `T`.

Effects: For all index positions (`idx0`, `idx1`, `idx2`) in `*this`'s domain, `this->put(idx0, idx1, idx2, this->get(idx0, idx1, idx2) ^ t.get(idx0, idx1, idx2))`.

Returns: `*this`.

6.6.5 Transpositions

[view.tensor.transpose]

1 These functions return transpositions of a *Tensor* object.

```
template <dimension_type D0,
          dimension_type D1,
          dimension_type D2>
typename transpose_view<D0, D1, D2>::const_type
transpose() const VSIP_NOTHROW;
```

Requires: `D0 < 3 && D1 < 3 && D2 < 3`.

Returns: A subview of the *Tensor* object containing a transposition of the specified dimensions. The *z*, *y*, and *x* dimensions are numbered 0, 1, and 2, respectively. `D0` should indicate dimension 0 of the transposed subview. `D1` and `D2` are similar.

Note: It is legal for `D0 == 0 && D1 = 1 && D2 = 2`, but this yields a subview equal to `*this`.

```
template <dimension_type D0,
          dimension_type D1,
          dimension_type D2>
typename transpose_view<D0, D1, D2>::type
transpose() VSIP_NOTHROW;
```

Requires: `*this` must be a *Tensor* object. `D0 < 3 && D1 < 3 && D2 < 3`.

Returns: A subview of the *Tensor* object containing a transposition of the specified dimensions. The *z*, *y*, and *x* dimensions are numbered 0, 1, and 2, respectively. `D0` should indicate dimension 0 of the transposed subview. `D1` and `D2` are similar.

Note: It is legal for `D0 == 0 && D1 = 1 && D2 = 2`, but this yields a subview equal to `*this`.

6.6.6 Value accessors

[view.tensor.valaccess]

[Note: The restrictions for `get` and `put` follow from the view requirements in [view.view]. Only additional restrictions occur here.]

```
reference_type
operator () (index_type idx0, index_type idx1, index_type
idx2) VSIP_NOTHROW;
```

Requires: This value accessor is provided only by `Tensor`, not `const_Tensor`. $idx0 < this->size(0)$, $idx1 < this->size(1)$, and $idx2 < this->size(2)$. An implementation should provide this accessor but is not required to do so.

Returns: Value at the `Index<3>` position $(idx0, idx1, idx2)$.

Note: A VSIPL++ Library implemented using a VSIPL implementation cannot provide this accessor because VSIPL does not provide such functionality.

```
const_reference_type
operator () (index_type idx0, index_type idx1,
index_type idx2) const VSIP_NOTHROW;
```

Requires: $idx0 < size(0)$, $idx1 < size(1)$, and $idx2 < size(2)$. An implementation should provide this accessor but is not required to do so.

Returns: Value at the `Index<3>` position $(idx0, idx1, idx2)$.

Note: A VSIPL++ Library implemented using a VSIPL implementation can provide this accessor but not its non-const version.

6.6.7 Subviews

[view.tensor.subviews]

- 1 These functions return subviews of a `const_Tensor` or `Tensor` object.

```
const_subview_type
get (Domain<3> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: `d` is a subset of the `Tensor`'s domain.

Returns: A subview object of `*this` with domain `Domain<3>(d[0].size(), d[1].size(), d[2].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
subview_type
```

```
operator () (Domain<3> const& d) VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. *d* is a subset of the Tensor's domain.

Returns: A subview Tensor object of **this* with domain `Domain<3>(d[0].size(), d[1].size(), d[2].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_subview_type
```

```
operator () (Domain<3> const& d) const VSIP_THROW((std::bad_alloc));
```

Requires: *d* is a subset of the *Tensor*'s domain.

Returns: A subview object of **this* with domain `Domain<3>(d[0].size(), d[1].size(), d[2].size())`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 1>::type
```

```
operator () (
    index_type      idx0,
    index_type      idx1,
    whole_domain_type)
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. `idx0 < this->size(0)`. `idx1 < this->size(1)`.

Returns: An *x* subview Vector object of **this* with the whole domain of `Domain<1>(this->size(2))`, fixed *z* value `idx0`, and fixed *y* value `idx1`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 1>::const_type
```

```
operator () (
    index_type      idx0,
    index_type      idx1,
    whole_domain_type)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx0 < this->size(0)` . `idx1 < this->size(1)` .

Returns: An *x* subview `const_Vector` object of `*this` with the whole domain of `Domain<1>(this->size(2))` , fixed *z* value `idx0`, and fixed *y* value `idx1` .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 2>::type
operator () (
    index_type          idx0,
    whole_domain_type,
    index_type          idx2)
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a Tensor object. `idx0 < this->size(0)` .
`idx2 < this->size(2)` .

Returns: An *y* subview `Vector` object of `*this` with the whole domain of `Domain<1>(this->size(1))` , fixed *z* value `idx0`, and fixed *x* value `idx2` .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 2>::const_type
operator () (
    index_type          idx0,
    whole_domain_type,
    index_type          idx2)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx0 < this->size(0)` . `idx2 < this->size(2)` .

Returns: An *y* subview `const_Vector` object of `*this` with the whole domain of `Domain<1>(this->size(1))` , fixed *z* value `idx0`, and fixed *x* value `idx2` .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<1, 2>::type
operator () (
    whole_domain_type,
```

```

    index_type      idx1,
    index_type      idx2)
VSIP_THROW((std::bad_alloc));

```

Requires: **this* must be a Tensor object. $idx1 < this->size(1)$.
 $idx2 < this->size(2)$.

Returns: An z subview Vector object of **this* with the whole domain of $Domain<1>(this->size(0))$, fixed y value $idx1$, and fixed x value $idx2$.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```

typename subvector<1, 2>::const_type

```

```

operator () (
    whole_domain_type,
    index_type      idx1,
    index_type      idx2)
const VSIP_THROW((std::bad_alloc));

```

Requires: $idx1 < this->size(1)$. $idx2 < this->size(2)$.

Returns: An z subview `const_Vector` object of **this* with the whole domain of $Domain<1>(this->size(0))$, fixed y value $idx1$, and fixed x value $idx2$.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```

typename subvector<0, 1>::subset_type

```

```

operator () (
    index_type      idx0,
    index_type      idx1,
    Domain<1> const& d)
VSIP_THROW((std::bad_alloc));

```

Requires: **this* must be a Tensor object. $idx0 < this->size(0)$. $idx1 < this->size(1)$. d is a subdomain of $Domain<1>(this->size(2))$.

Returns: An x subview Vector object of **this* object such that its domain $Domain<1>(d.size())$ is the subdomain specified by d , has fixed z value $idx0$, and has fixed y value $idx1$.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 1>::const_subset_type  
operator () (  
    index_type          idx0,  
    index_type          idx1,  
    Domain<1> const& d)  
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx0 < this->size(0)` . `idx1 < this->size(1)` . `d` is a subdomain of `Domain<1>(this->size(2))` .

Returns: An *x* subview object of **this* such that its domain `Domain<1>(d.size())` is the subdomain specified by `d`, has fixed *z* value `idx0`, and has fixed *y* value `idx1` .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 2>::subset_type  
operator () (  
    index_type          idx0,  
    Domain<1> const& d,  
    index_type          idx2)  
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. `idx0 < this->size(0)` . `d` is a subdomain of `Domain<1>(this->size(1))` . `idx2 < this->size(2)` .

Returns: An *y* subview Vector object of **this* such that its domain `Domain<1>(d.size())` is the subdomain specified by `d`, has fixed *z* value `idx0`, and has fixed *x* value `idx2` .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<0, 2>::const_subset_type  
operator () (  
    index_type          idx0,  
    Domain<1> const& d,  
    index_type          idx2)  
const VSIP_THROW((std::bad_alloc));
```

Requires: $\text{idx0} < \text{this->size}(0)$. d is a subdomain of $\text{Domain}\langle 1 \rangle(\text{this->size}(1))$. $\text{idx2} < \text{this->size}(2)$.

Returns: An y subview object of $*\text{this}$ such that its domain $\text{Domain}\langle 1 \rangle(d.\text{size}())$ is the subdomain specified by d , has fixed z value idx0 , and has fixed x value idx2 .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<1, 2>::subset_type
```

```
operator () (
    Domain<1> const& d,
    index_type      idx1,
    index_type      idx2)
VSIP_THROW((std::bad_alloc));
```

Requires: $*\text{this}$ must be a Tensor object. d is a subdomain of $\text{Domain}\langle 1 \rangle(\text{this->size}(0))$. $\text{idx1} < \text{this->size}(1)$. $\text{idx2} < \text{this->size}(2)$.

Returns: An z subview Vector object of t such that its domain $\text{Domain}\langle 1 \rangle(d.\text{size}())$ is the subdomain specified by d , has fixed y value idx1 , and has fixed x value idx2 .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename subvector<1, 2>::const_subset_type
```

```
operator () (
    Domain<1> const& d,
    index_type      idx1,
    index_type      idx2)
const VSIP_THROW((std::bad_alloc));
```

Requires: d is a subdomain of $\text{Domain}\langle 1 \rangle(\text{this->size}(0))$. $\text{idx1} < \text{this->size}(1)$. $\text{idx2} < \text{this->size}(2)$.

Returns: An z subview object of t such that its domain $\text{Domain}\langle 1 \rangle(d.\text{size}())$ is the subdomain specified by d , has fixed y value idx1 , and has fixed x value idx2 .

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<0>::type
operator() (
    index_type          idx0,
    whole_domain_type,
    whole_domain_type)
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. `idx0 < this->size(0)`.

Returns: An *y-x* subview Matrix object of **this* with the whole domain of `Domain<2>(this->size(1), this->size(2))` and fixed *z* value `idx0`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<0>::const_type
operator() (
    index_type          idx0,
    whole_domain_type,
    whole_domain_type)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx0 < this->size(0)`.

Returns: An *y-x* subview const_Matrix object of **this* with the whole domain of `Domain<2>(this->size(1), this->size(2))` and fixed *z* value `idx0`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<1>::type
operator() (
    whole_domain_type,
    index_type          idx1,
    whole_domain_type)
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. `idx1 < this->size(1)`.

Returns: An *z-x* subview Matrix object of **this* with the whole domain of `Domain<2>(this->size(0), this->size(2))` and fixed *y* value `idx1`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<1>::const_type
operator () (
    whole_domain_type,
    index_type      idx1,
    whole_domain_type)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx1 < this->size(1)`.

Returns: An z - x subview `const_Matrix` object of `*this` with the whole domain of `Domain<2>(this->size(0), this->size(2))` and fixed y value `idx1`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<2>::type
operator () (
    whole_domain_type,
    whole_domain_type,
    index_type      idx2)
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Tensor` object. `idx2 < this->size(2)`.

Returns: An z - y subview `Matrix` object of `*this` with the whole domain of `Domain<2>(this->size(0), this->size(1))` and fixed x value `idx2`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<2>::const_type
operator () (
    whole_domain_type,
    whole_domain_type,
    index_type      idx2)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx2 < this->size(2)`.

Returns: An z - y subview `const_Matrix` object of `*this` with the whole domain of `Domain<2>(this->size(0), this->size(1))` and fixed x value `idx2`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<0>::subset_type
operator () (
    index_type          idx0,
    Domain<1> const& d1,
    Domain<1> const& d2)
VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Tensor` object. `idx0 < this->size(0)`. `d1` is a subdomain of `Domain<1>(this->size(1))`. `d2` is a subdomain of `Domain<1>(this->size(2))`.

Returns: A y - x subview `Matrix` object of `*this` such that its domain `Domain<2>(d1.size(), d2.size())` is the subdomain specified by `Domain<2>(d1, d2)` and has fixed z value `idx0`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<0>::const_subset_type
operator () (
    index_type          idx0,
    Domain<1> const& d1,
    Domain<1> const& d2)
const VSIP_THROW((std::bad_alloc));
```

Requires: `idx0 < this->size(0)`. `d1` is a subdomain of `Domain<1>(this->size(1))`. `d2` is a subdomain of `Domain<1>(this->size(2))`.

Returns: A y - x subview object of `*this` such that its domain `Domain<2>(d1.size(), d2.size())` is the subdomain specified by `Domain<2>(d1, d2)` and has fixed z value `idx0`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<1>::subset_type
```

```
operator() (
    Domain<1> const& d0,
    index_type      idx1,
    Domain<1> const& d2)
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. *d0* is a subdomain of `Domain<1>(this->size(0)).idx1 < this->size(1)`. *d2* is a subdomain of `this->size(2)`.

Returns: A *z-x* subview Matrix object of **this* such that its domain `Domain<2>(d0.size(), d1.size())` is the subdomain specified by `Domain<2>(d0, d2)` and has fixed *y* value *idx1*.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<1>::const_subset_type
```

```
operator() (
    Domain<1> const& d0,
    index_type      idx1,
    Domain<1> const& d2)
const VSIP_THROW((std::bad_alloc));
```

Requires: *d0* is a subdomain of `Domain<1>(this->size(0)).idx1 < this->size(1)`. *d2* is a subdomain `Domain<2>(d0.size(), d2.size())` of `Domain<1>(this->size(2))`.

Returns: A *z-x* subview object of **this* such that its domain is the subdomain specified by `Domain<2>(d0, d2)` and has fixed *y* value *idx1*.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<2>::subset_type
```

```
operator() (
    Domain<1> const& d0,
    Domain<1> const& d1,
    index_type      idx2)
VSIP_THROW((std::bad_alloc));
```

Requires: **this* must be a Tensor object. *d0* is a subdomain of `Domain<1>(this->size(0))`. *d1* is a subdomain of `Domain<1>(this->size(1)).idx2 < this->size(2)`.

Returns: A z - y subview `Matrix` object of `*this` such that its domain `Domain<2>(d0.size(), d1.size())` is the subdomain specified by `Domain<2>(d0, d1)` and has fixed x value `idx2`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
typename submatrix<2>::const_subset_type
operator() (
    Domain<1> const& d0,
    Domain<1> const& d1,
    index_type      idx2)
const VSIP_THROW((std::bad_alloc));
```

Requires: `d0` is a subdomain of `Domain<1>(this->size(0))`. `d1` is a subdomain of `Domain<1>(this->size(1))`. `idx2 < this->size(2)`.

Returns: A z - y subview object of `*this` such that its domain `Domain<2>(d0.size(), d1.size())` is the subdomain specified by `Domain<2>(d0, d1)` and has fixed x value `idx2`.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_realview_type
real() const VSIP_THROW((std::bad_alloc));
```

Requires: `T` must be a complex type.

Returns: A subview `const_Tensor` object of `*this` with the same domain but accessing only the real parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
realview_type
real() VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Tensor` object. `T` must be a complex type.

Returns: A subview `Tensor` object of `*this` with the same domain but accessing only the real parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
const_imagview_type
imag() const VSIP_THROW((std::bad_alloc));
```

Requires: T must be a complex type.

Returns: A subview `const_Tensor` object of `*this` with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

```
imagview_type
imag() VSIP_THROW((std::bad_alloc));
```

Requires: `*this` must be a `Tensor` object. T must be a complex type.

Returns: A subview `Tensor` object of `*this` with the same domain but accessing only the imaginary parts of the complex values of the object.

Throws: `std::bad_alloc` indicating memory allocation for the underlying block failed.

6.6.8 Accessors

[view.tensor.accessors]

```
block_type&
block() const VSIP_NOTHROW;
```

Returns: The tensor's underlying block. `const_Tensor::block()` returns a `block_type const&` while `Tensor::block()` returns a `block_type &`.

```
length_type
size() const VSIP_NOTHROW;
```

Returns: The number of values in the tensor.

```
length_type
size(dimension_type d) const VSIP_NOTHROW;
```

Requires: $0 \leq d \leq 2$.

Returns: The number of values in the specified dimension of the tensor.

6.6.9 *Tensor type conversion*

[**view.tensor.convert**]

- 1 [*Note:* The class `ViewConversion` converts between constant and non-constant view classes.]
- 2 For type `View` equal to `Tensor` or to `const_Tensor`, `ViewConversion<View, T, Block>::const_view_type` equals `const_Tensor<T, Block>` and `ViewConversion<View, T, Block>::view_type` equals `Tensor<T, Block>`.

- 1 This clause defines complex numbers, which are composed of real and imaginary parts.
- 2 (ISO14882, [lib.complex.numbers]) is incorporated by reference. These classes and functions are incorporated into the `vsip` namespace. [Example: `vsip::complex` and `vsip::polar` are synonyms for the class template `std::complex` and the function `std::polar`.]

Header <vsip/complex.hpp> synopsis

```
namespace vsip {
    using std::complex;
    using std::polar;

    /* recttopolar */
    template <typename T1,
              typename T2>
    void recttopolar(complex<T1> const, T2&, T2&)
        VSIP_NOTHROW;

    template <typename T1,
              typename T2,
              template <typename> class const_View,
              typename Block0,
              typename Block1,
              typename Block2>
    void recttopolar(const_View<complex<T1>, Block0>,
                    View<T2, Block1>, View<T2, Block2>)
        VSIP_NOTHROW;

    /* polartorect */
    template <typename T>
    complex<T> polartorect(T const& rho, T const& theta = 0)
        VSIP_NOTHROW;
}
```

7.1. [COMPLEX.CONVERT] CONVERSIONS

```
template <typename T,
          typename Block0,
          typename Block1>
const_View<complex<T>, unspecified>
    polartorect(const_View<T, Block0>)
    VSIP_NOTHROW;
template <typename T,
          typename Block0,
          typename Block1>
const_View<complex<T>, unspecified>
    polartorect(const_View<T, Block0>, const_View<T, Block1>)
    VSIP_NOTHROW;
}
```

7.1 Conversions

[**complex.convert**]

- 1 [Note: The functionality of the VSIPL functions `vsip_polar_f`, `vsip_vpolar_f`, and `vsip_mpolar_f`, which produce magnitude and phase angle values from given complex values, is provided by `recttopolar`. This function is so named to avoid conflict with the VSIPL++ function `polar`, which, as specified in (ISO14882, [lib.complex.value.ops]), constructs a complex number given a magnitude and phase angle.]
- 2 [Note: The VSIPL++ equivalents of VSIPL functions `vsip_rect_f`, `vsip_vrect_f`, and `vsip_mrect_f`, which produce complex numbers given magnitude and phase angle values, are `polartorect` and its synonym `polar`.]

```
template <typename T1,
          typename T2>
void
recttopolar(complex<T1> const z, T2& rho, T2& theta) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported has T1 and T2 both the same as `scalar_f`. An implementation is permitted to prevent instantiation for other choices of T1 and T2 . T1 must be assignable to T2 .
(z.real() != 0) || (z.imag() != 0).

Effects: Stores the polar representation in `rho` and `theta`. `rho` contains the radius, and `theta` contains the phase argument.

Note: Some specializations correspond to VSIPL function `vsip_polar_f`.

```

template <typename                T1,
         typename                T2,
         template <typename, typename> class const_View,
         template <typename, typename> class View,
         typename                Block0,
         typename                Block1,
         typename                Block2>
void
recttopolar (
    const_View<complex<T1>, Block0>& z,
    View<T2, Block1> rho,
    View<T2, Block2> theta)
VSIP_NOTHROW;

```

Requires: `z`, `rho`, and `theta` must be element conformant. The only specializations which must be supported have `T1` and `T2` both the same as `scalar_f` and `const_View` and `View` the same as `const_Vector` and `Vector` or `const_Matrix` and `Matrix`, respectively. An implementation is permitted to prevent instantiation for other choices of `T1`, `T2`, and `View`. `T1` must be assignable to `T2`. For no value `val` in `z`, `val.real() == 0 && val.imag() == 0`.

Effects: Stores the polar representations of the complex numbers in `z` in `rho` and `theta`. `rho` contains the radii, and `theta` contains the phase arguments such that, for corresponding values `val_z`, `val_rho`, and `val_theta` and `recttopolar(val_z, val_rho, val_theta)`, `val_rho == val_rho` && `val_theta == val_theta`.

Note: Some specializations correspond to VSIP functions `vsip_vpolar_f` and `vsip_mpolar_f`.

```

template <typename T>
complex<T>
polartorect(T const& rho, T const& theta = 0) VSIP_NOTHROW;

```

Requires: The only specialization which must be supported has `T` the same as `scalar_f`. An implementation is permitted to prevent instantiation of `polartorect<T>` for other choices of `T`.

Returns: `vsip::polar(rho, theta)`.

Note: Some specializations correspond to VSIP function `vsip_rect_f`.

```

template <typename                T,

```

```
        template <typename, typename> class const_View,  
        typename                               Block0,  
        typename                               Block1>  
const_View<complex<T>, unspecified>  
polartorect(const_View<T, Block0> rho)  
    VSIP_NOTHROW;
```

Requires: The only specialization which must be supported has T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation of `polartorect<T>` for other choices of T and `const_View`.

Returns: A `const_View` `v` such that, for every value `val_v` in `v`, `val_v == polartorect(val_rho, 0)` where `val_rho` is the corresponding value in `rho`, respectively.

Note: Some specializations correspond to VSIP functions `vsip_vrect_f` and `vsip_mrect_f`.

```
        template <typename                               T,  
        template <typename, typename> class const_View,  
        typename                               Block0,  
        typename                               Block1>  
const_View<complex<T>, unspecified>  
polartorect (  
    const_View<T, Block0> rho,  
    const_View<T, Block1> theta)  
    VSIP_NOTHROW;
```

Requires: `rho` and `theta` must be element conformant. The only specialization which must be supported has T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation of `polartorect<T>` for other choices of T and `const_View`.

Returns: A `const_View` `v` such that, for every value `val_v` in `v`, `val_v == polartorect(val_rho, val_theta)` where `val_rho` and `val_theta` are the corresponding values in `rho` and `theta`, respectively.

Note: Some specializations correspond to VSIP functions `vsip_vrect_f` and `vsip_mrect_f`.

-
- 1 VSIP++ supports a wide variety of mathematical C++ functions. Many of these can be applied to both scalars and views. For example, applying `sin` to a floating point number yields its sine. Applying it to a one-dimensional view `v` yields another one-dimensional view of the same size with values equal to the sines of corresponding `v` values.
 - 2 `<vsip/math.hpp>` contains all declarations in this clause unless otherwise indicated.

8.1 Enumerations

[math.enum]

```

namespace vsip {
    // enumerations
    enum mat_op_type { mat_ntrans, mat_trans, mat_herm, mat_conj };
    enum product_side_type { mat_lside, mat_rside };
    enum storage_type { qrd_nosaveq, qrd_saveq1, qrd_saveq,
                       svd_uvnos, svd_uvpart, svd_uvfull };
}

```

- 1 *[Note: The `mat_op_type` enumerated value `mat_ntrans` indicates the matrix should not be transposed. `mat_trans` indicates matrix transpose. `mat_herm` indicates the Hermitian transpose or conjugate transpose. `mat_conj` indicates the complex conjugate of matrix entries should occur.]*
- 2 *[Note: `enum product_side_type` indicates whether to use left or right multiplication in matrix products.]*
- 3 *[Note: `enum storage_type` indicates the storage format for decomposed matrixes. Constants appropriate for QR decomposition begin with a `qrd` prefix. See [\[math.solvers.qr\]](#) for an explanation of these constants. Constants appropriate for singular value decomposition begin with an `svd` prefix. See [\[math.solvers.svd\]](#) for an explanation of these constants.]*

- 1 A C++ function f on a scalar can be *extended element-wise* to a view by applying f to each value in the view. That is, a function f operating on a view v yields another view w such that, for all $\text{Index}<d>\text{es } (i_1, \dots, i_d)$ in v 's domain, a corresponding $\text{Index}<d> (j_1, \dots, j_d)$ is in w 's domain and $w(j_1, \dots, j_d) == f(v(i_1, \dots, i_d))$. If val has $v::\text{value_type}$ type and $f(val)$ is a valid C++ expression and yields a value with type t , then $w::\text{value_type}$ is t . Element-wise extensions to more than one view are analogously defined.
- 2 A binary C++ function f on scalars can be *extended element-wise* to a function on a view and a scalar by applying f to each pair of a value in the view and the scalar. That is, a function f operating on a view v and a scalar s yields another view w such that, for all $\text{Index}<d>\text{es } (i_1, \dots, i_d)$ in v 's domain, a corresponding $\text{Index}<d> (j_1, \dots, j_d)$ is in w 's domain and $w(j_1, \dots, j_d) = f(v(i_1, \dots, i_d), s)$. If val has $v::\text{value_type}$ type and $f(val, s)$ is a valid C++ expression and yields a value with type t , then $w::\text{value_type}$ is t .
- 3 [*Example:* `max` can be extended element-wise to a function on a one-dimensional vector and a scalar. Given a `Vector` v containing the index-value pairs $(0, 0)$, $(1, 4)$, $(2, -2)$, $(3, -20)$, `max(v, 0)` is a `Vector` containing $(0, 0)$, $(1, 4)$, $(2, 0)$, $(3, 0)$.]
- 4 A binary C++ function f on scalars can be *extended element-wise* to a function on a scalar and a view and is defined analogously to the element-wise extension of a binary function on a view and a scalar.
- 5 [*Note:* Extending C++ functions to views uses the concept of conformance in two different ways. A unary function's element-wise extension yields a view element-conformant to its input view if the output view has one value for each input value. When a function takes multiple input operands, frequently they must be element-conformant. For example, element-wise addition (+) is restricted to element-conformant views because these views have corresponding values that can be added using the scalar addition function.]

- 1 This subclause defines common mathematical functions on scalars and views.

```

namespace vsip {
  // type promotion traits
  template <typename, typename>
  class Promotion;

  // scalar functions and element-wise extensions
  // [math.fns.scalar] and [math.fns.elementwise]
  /* acos */
  template <typename T>
  T
  acos(T) VSIP_NOTHROW;
  template <typename T,
    template <typename, typename> class const_View,
    typename Block>
  const_View<T, unspecified>
  acos(const_View<T, Block>) VSIP_NOTHROW;

  /* add, + */
  template <typename T1, typename T2>
  typename Promotion<T1, T2>::type
  add(T1, T2) VSIP_NOTHROW;
  template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
  const_View<typename Promotion<T1, T2>::type, unspecified>
  add(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

  template <typename T1, typename T2>
  typename Promotion<T1, T2>::type
  operator+(T1, T2) VSIP_NOTHROW;
  template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
  const_View<typename Promotion<T1, T2>::type, unspecified>
  operator+(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

  /* am */
  template <typename T1, typename T2, typename T3,
    template <typename, typename> class const_View,
    typename Block1, typename Block2, typename Block3>
  const_View<typename Promotion<T1,
    typename Promotion<T2, T3>::type>::type,
    unspecified>
  am(const_View<T1, Block1>,
    const_View<T2, Block2>,
    const_View<T3, Block3>) VSIP_NOTHROW;

```

```

/* land, band, &&, & */
template <template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<bool, unspecified>
land(const_View<bool, Block1>,
      const_View<bool, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
band(const_View<T1, Block1>,
      const_View<T2, Block2>) VSIP_NOTHROW;

template <template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<bool, unspecified>
operator&&(const_View<bool, Block1>,
           const_View<bool, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator&(const_View<T1, Block1>,
           const_View<T2, Block2>) VSIP_NOTHROW;

/* arg */
template <typename T>
T
arg(complex<T> const&) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
arg(const_View<complex<T>, Block>) VSIP_NOTHROW;

/* asin */
template <typename T>
T
asin(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
asin(const_View<T, Block>) VSIP_NOTHROW;

/* atan, atan2 */
template <typename T>

```

```

T
atan(T) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
atan(const_View<T, Block>) VSIP_NOTHROW;

template <typename T1, typename T2>
typename Promotion<T1, T2>::type
atan2(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
atan2(const_View<T1, Block1>,
      const_View<T2, Block2>) VSIP_NOTHROW;

/* ceil */
template <typename T>
T
ceil(T) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
ceil(const_View<T, Block>) VSIP_NOTHROW;

/* cmplx */
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<complex<typename Promotion<T1, T2>::type>,
          unspecified>
cmplx(const_View<T1, Block1>,
      const_View<T2, Block2>) VSIP_NOTHROW;

/* conj */
template <typename T>
complex<T>
conj(complex<T> const&) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<complex<T>, unspecified>
conj(const_View<complex<T>, Block>) VSIP_NOTHROW;

/* cos */
template <typename T>

```

```

T
cos(T) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>
cos(const_View<T, Block>) VSIP_NOTHROW;

/* cosh */
template <typename T>
T
cosh(T) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>
cosh(const_View<T, Block>) VSIP_NOTHROW;

/* div, / */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
div(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
div(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2>
typename Promotion<T1, T2>::type
operator/(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator/(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* eq, == */
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<bool, unspecified>
eq(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,

```

```

    typename Block1, typename Block2>
    const_View<bool, unspecified>
    operator==(const_View<T1, Block1>,
               const_View<T2, Block2>) VSIP_NOTHROW;

/* euler */
template <typename T,
          template <typename, typename> class const_View,
          typename Block>
const_View<complex<T>, unspecified>
euler(const_View<T, Block>) VSIP_NOTHROW;

/* exp, exp10 */
template <typename T>
T
exp(T) VSIP_NOTHROW;
template <typename T,
          template <typename, typename> class const_View,
          typename Block>
const_View<T, unspecified>
exp(const_View<T, Block>) VSIP_NOTHROW;

template <typename T>
T
exp10(T) VSIP_NOTHROW;
template <typename T,
          template <typename, typename> class const_View,
          typename Block>
const_View<T, unspecified>
exp10(const_View<T, Block>) VSIP_NOTHROW;

/* floor */
template <typename T>
T
floor(T) VSIP_NOTHROW;
template <typename T,
          template <typename, typename> class const_View,
          typename Block>
const_View<T, unspecified>
floor(const_View<T, Block>) VSIP_NOTHROW;

/* fmod */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
fmod(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>

```

```

fmod(const_View<T1, Block1>,
      const_View<T2, Block2>) VSIP_NOTHROW;

/* ge, >= */
template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<bool, unspecified>
ge(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<bool, unspecified>
operator>=(const_View<T1, Block1>,
           const_View<T2, Block2>) VSIP_NOTHROW;

/* gt, > */
template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<bool, unspecified>
gt(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<bool, unspecified>
operator>(const_View<T1, Block1>,
          const_View<T2, Block2>) VSIP_NOTHROW;

/* hypot */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
hypot(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
          template <typename, typename> class const_View,
          typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
hypot(const_View<T1, Block1>,
      const_View<T2, Block2>) VSIP_NOTHROW;

/* imag */
template <typename T>
T
imag(complex<T> const&) VSIP_NOTHROW;
template <typename T,

```

```

    template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
imag(const_View<complex<T>, Block>) VSIP_NOTHROW;

/* jmul */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
jmul(T1, T2)
    VSIP_NOTHROW;
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
jmul(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* le, <= */
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<bool, unspecified>
le(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<bool, unspecified>
operator<=(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* log, log10 */
template <typename T>
T
log(T) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>
log(const_View<T, Block>) VSIP_NOTHROW;

template <typename T>
T
log10(T) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>

```

```

log10(const_View<T, Block>) VSIP_NOTHROW;

/* lt, < */
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<bool, unspecified>
lt(const_View<T1, Block1>,
   const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<bool, unspecified>
operator<(const_View<T1, Block1>,
         const_View<T2, Block2>) VSIP_NOTHROW;

/* ma */
template <typename T1, typename T2, typename T3,
         template <typename, typename> class const_View,
         typename Block1, typename Block2, typename Block3>
const_View<typename Promotion<T1,
                             typename Promotion<T2, T3>::type>::type,
          unspecified>
ma(const_View<T1, Block1>,
   const_View<T2, Block2>,
   const_View<T3, Block3>) VSIP_NOTHROW;

/* mag, magsq */
template <typename T>
T
mag(complex<T> const&) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
mag(const_View<complex<T>, Block>) VSIP_NOTHROW;

template <typename T>
T
mag(T) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
mag(const_View<T, Block>) VSIP_NOTHROW;

template <typename T>
T

```

```

magsq(complex<T> const&) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>
magsq(const_View<complex<T>, Block>) VSIP_NOTHROW;

/* max, maxmg, maxmgsq */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
max(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
max(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
maxmg(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
maxmgsq(const_View<complex<T1>, Block1>,
    const_View<complex<T2>, Block2>) VSIP_NOTHROW;

/* min, minmg, minmgsq */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
min(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
min(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
minmg(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

```

```

template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
minmgsq(const_View<complex<T1>, Block1>,
        const_View<complex<T2>, Block2>)
    VSIP_NOTHROW;

/* msb */
template <typename T1, typename T2, typename T3,
         template <typename, typename> class const_View,
         typename Block1, typename Block2, typename Block3>
const_View<typename Promotion<T1,
        typename Promotion<T2, T3>::type>::type,
        unspecified>
msb(const_View<T1, Block1>,
    const_View<T2, Block2>,
    const_View<T3, Block3>) VSIP_NOTHROW;

/* mul, * */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
mul(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
mul(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2>
typename Promotion<T1, T2>::type
operator*(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator*(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* ne, != */
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<bool, unspecified>
ne(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

```

```

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<bool, unspecified>
operator!=(const_View<T1, Block1>,
          const_View<T2, Block2>) VSIP_NOTHROW;

/* neg, - */
template <typename T>
T
neg(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
neg(const_View<T, Block>) VSIP_NOTHROW;

template <typename T>
T
operator-(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
operator-(const_View<T, Block>) VSIP_NOTHROW;

/* lnot, bnot, !, ~ */
template <template <typename, typename> class const_View,
        typename Block>
const_View<bool, unspecified>
lnot(const_View<bool, Block>) VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
bnot(const_View<T, Block>) VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<bool, unspecified>
operator!(const_View<bool, Block>) VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<bool, unspecified>
operator~(const_View<T, Block>) VSIP_NOTHROW;

```

```

/* lor, bor, ||, | */
template <template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<bool, unspecified>
lor(const_View<bool, Block1>,
    const_View<bool, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
bor(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<bool, unspecified>
operator||(const_View<bool, Block1>,
    const_View<bool, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator|(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* pow */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
pow(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
pow(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

/* real */
template <typename T>
T
real(complex<T> const&) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
real(const_View<complex<T>, Block>) VSIP_NOTHROW;

```

```

/* recip */
template <typename T>
T
recip(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
recip(const_View<T, Block>) VSIP_NOTHROW;

/* rsqrt */
template <typename T>
T
rsqrt(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
rsqrt(const_View<T, Block>) VSIP_NOTHROW;

/* sbm */
template <typename T1, typename T2, typename T3,
        template <typename, typename> class const_View,
        typename Block1, typename Block2, typename Block3>
const_View<typename Promotion<T1,
        typename Promotion<T2, T3>::type>::type,
        unspecified>
sbm(const_View<T1, Block1>,
    const_View<T2, Block2>,
    const_View<T3, Block3>) VSIP_NOTHROW;

/* sin */
template <typename T>
T
sin(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>
sin(const_View<T, Block>) VSIP_NOTHROW;

/* sinh */
template <typename T>
T
sinh(T) VSIP_NOTHROW;
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
const_View<T, unspecified>

```

```

sinh(const_View<T, Block>) VSIP_NOTHROW;

/* sq */
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
sq(const_View<T, Block>) VSIP_NOTHROW;

/* sqrt */
template <typename T>
T
sqrt(T) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
sqrt(const_View<T, Block>) VSIP_NOTHROW;

/* sub, - */
template <typename T1, typename T2>
typename Promotion<T1, T2>::type
sub(T1, T2) VSIP_NOTHROW;
template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
sub(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
         template <typename, typename> class const_View,
         typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator-(const_View<T1, Block1>,
          const_View<T2, Block2>) VSIP_NOTHROW;

/* tan */
template <typename T>
T
tan(T) VSIP_NOTHROW;
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
const_View<T, unspecified>
tan(const_View<T, Block>) VSIP_NOTHROW;

/* tanh */
template <typename T>

```

```

T
tanh(T) VSIP_NOTHROW;
template <typename T,
    template <typename, typename> class const_View,
    typename Block>
const_View<T, unspecified>
tanh(const_View<T, Block>) VSIP_NOTHROW;

/* lxor, bxor, ^ */
template <template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<bool, unspecified>
lxor(const_View<bool, Block1>,
    const_View<bool, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
bxor(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator^(const_View<T1, Block1>,
    const_View<T2, Block2>) VSIP_NOTHROW;

// element-wise extensions with scalars
// [math.fns.scalarview]
/* add, + */
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
add(T1, const_View<T2, Block>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator+(T1, const_View<T2, Block>) VSIP_NOTHROW;

/* am */
template <typename T1, typename T2, typename T3,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1, typename

```

```

        Promotion<T2, T3>::type>::type,
        unspecified>
am(const_View<T1, Block1>,
    T2,
    const_View<T3, Block2>) VSIP_NOTHROW;

/* div, / */
template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
div(T1,
    const_View<T2, Block>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
div(const_View<T1, Block>,
    T2) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator/(T1,
    const_View<T2, Block>) VSIP_NOTHROW;

template <typename T1, typename T2,
        template <typename, typename> class const_View,
        typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator/(const_View<T1, Block>,
    T2) VSIP_NOTHROW;

/* expoavg */
template <typename T1, typename T2, typename T3,
        template <typename, typename> class const_View,
        typename Block1, typename Block2>
const_View<typename Promotion<T1,
        typename Promotion<T2, T3>::type>::type,
        unspecified>
expoavg(T1,
    const_View<T2, Block1>,
    const_View<T3, Block2>) VSIP_NOTHROW;

/* ma */
template <typename T1, typename T2, typename T3,
        template <typename, typename> class const_View,

```

```

    typename Block>
const_View<typename Promotion<T1,
    typename Promotion<T2, T3>::type>::type,
    unspecified>
ma(const_View<T1, Block>,
    T2,
    T3) VSIP_NOTHROW;
template <typename T1, typename T2, typename T3,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1,
    typename Promotion<T2, T3>::type>::type,
    unspecified>
ma(const_View<T1, Block1>,
    T2,
    const_View<T3, Block2>) VSIP_NOTHROW;
template <typename T1, typename T2, typename T3,
    template <typename, typename> class const_View,
    typename Block1, typename Block2>
const_View<typename Promotion<T1,
    typename Promotion<T2, T3>::type>::type,
    unspecified>
ma(const_View<T1, Block1>,
    const_View<T2, Block2>,
    T3) VSIP_NOTHROW;

/* mul, * */
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
mul(T1, const_View<T2, Block>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
operator*(T1, const_View<T2, Block>) VSIP_NOTHROW;

/* sub, - */
template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>
const_View<typename Promotion<T1, T2>::type, unspecified>
sub(T1, const_View<T2, Block>) VSIP_NOTHROW;

template <typename T1, typename T2,
    template <typename, typename> class const_View,
    typename Block>

```

```

const_View<typename Promotion<T1, T2>::type, unspecified>
operator-(T1, const_View<T2, Block>) VSIP_NOTHROW;

// Element-wise extensions of user-specified functions
// [math.fns.userelt]
/* unary */
template <typename OutputType,
          typename UnaryFunction,
          template <typename, typename> class const_View,
          typename InputType,
          typename Block>
const_View<OutputType, unspecified>
unary(UnaryFunction f,
      const_View<InputType, Block>);

template <typename OutputType,
          template <typename, typename> class const_View,
          typename InputType,
          typename Block>
const_View<OutputType, unspecified>
unary(OutputType (*)(InputType),
      const_View<InputType, Block>);

template <typename UnaryFunction,
          template <typename, typename> class const_View,
          typename Block>
const_View<typename UnaryFunction::result_type, unspecified>
unary(UnaryFunction,
      const_View<typename UnaryFunction::argument_type,
                Block>);

/* binary */
template <typename OutputType,
          typename BinaryFunction,
          template <typename, typename> class const_View,
          typename InputType0,
          typename Block0,
          typename InputType1,
          typename Block1>
const_View<OutputType, unspecified>
binary(BinaryFunction f,
       const_View<InputType0, Block0>,
       const_View<InputType1, Block1>);

template <typename OutputType,
          template <typename, typename> class const_View,
          typename InputType0,
          typename Block0,
          typename InputType1,
          typename Block1>

```

```

    typename Block1>
    const_View<OutputType, unspecified>
    binary(OutputType (*)(InputType0, InputType1),
           const_View<InputType0, Block0>,
           const_View<InputType1, Block1>);

template <typename BinaryFunction,
          template <typename, typename> class const_View,
          typename Block0,
          typename Block1>
const_View<typename BinaryFunction::result_type, unspecified>
binary(BinaryFunction,
       const_View<typename
                 BinaryFunction::first_argument_type,
                 Block0>,
       const_View<typename
                 BinaryFunction::second_argument_type,
                 Block1>);

/* ternary */
template <typename OutputType,
          typename TernaryFunction,
          template <typename, typename> class const_View,
          typename InputType0,
          typename Block0,
          typename InputType1,
          typename Block1,
          typename InputType2,
          typename Block2>
const_View<OutputType, unspecified>
ternary(TernaryFunction f,
       const_View<InputType0, Block0>,
       const_View<InputType1, Block1>,
       const_View<InputType2, Block2>);

template <typename OutputType,
          template <typename, typename> class const_View,
          typename InputType0,
          typename Block0,
          typename InputType1,
          typename Block1,
          typename InputType2,
          typename Block2>
const_View<OutputType, unspecified>
ternary(OutputType (*)(InputType0, InputType1, InputType2),
       const_View<InputType0, Block0>,
       const_View<InputType1, Block1>,
       const_View<InputType2, Block2>);

```

```

// Reduction functions
// [math.fns.reductions]
/* alltrue */
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
alltrue(const_View<T, Block>) VSIP_NOTHROW;

/* anytrue */
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
anytrue(const_View<T, Block>) VSIP_NOTHROW;

/* meanval, meansqval */
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
meanval(const_View<T, Block>) VSIP_NOTHROW;

template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
meansqval(const_View<T, Block>) VSIP_NOTHROW;

/* sumval, sumsqval */
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
sumval(const_View<T, Block>) VSIP_NOTHROW;

template <template <typename, typename> class const_View,
         typename Block>
length_type
sumval(const_View<bool, Block>) VSIP_NOTHROW;

template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
sumsqval(const_View<T, Block>) VSIP_NOTHROW;

// Reduction functions also returning indices
// [math.fns.reductidx]

```

```

/* maxmgsqval, maxmgval, maxval */
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
maxmgsqval(const_View<complex<T>, Block>,
           Index<View<complex<T>, Block>::dim>&)
    VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
maxmgval(const_View<complex<T>, Block>,
         Index<View<complex<T>, Block>::dim>&)
    VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
maxval(const_View<T, Block>,
       Index<View<T, Block>::dim>&)
    VSIP_NOTHROW;

/* minmgsqval, minmgval, minval */
template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
minmgsqval(const_View<complex<T>, Block>,
           Index<View<complex<T>, Block>::dim>&)
    VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
minmgval(const_View<complex<T>, Block>,
         Index<View<complex<T>, Block>::dim>&)
    VSIP_NOTHROW;

template <typename T,
        template <typename, typename> class const_View,
        typename Block>
T
minval(const_View<T, Block>,
       Index<View<T, Block>::dim>&)
    VSIP_NOTHROW;

```

```
}
```

8.3.1 *Type Promotions*

[math.fns.promotions]

- 1 The class `Promotion` implements standard arithmetic conversions in (ISO14882, [conv]). [*Note:* When operating on two views containing different value types `T1` and `T2`, `Promotion<T1, T2>::type` usually yields the resulting view's value type.] [*Example:* `Promotion<char, int>::type` is `int` since adding a `char` and an `int` yields an `int`. `Promotion` is not needed when operating on two values with the same type so, for all types `T`, `Promotion<T, T>::type` is `T`.]
- 2 An implementation supporting only operations on views specialized for particular value types need not define the `Promotion` template class, but all declarations using the template class must be equivalent to those if it was defined.
- 3 For all ordered pairs `T1` and `T2` of `bool`, integral types, and floating types, `Promotion<T1, T2>::type` is the type specified in (ISO14882, [conv]).
- 4 For all types `T`, `Promotion<T, T>::type` is `T`.
- 5 For all types `T1` and `T2`, `Promotion<complex<T1>, complex<T2>>::type` is `complex<typename Promotion<T1, T2>::type>`, `Promotion<complex<T1>, T2>::type` is `complex<Promotion<T1, T2>::type>`, and `Promotion<T1, complex<T2>>::type` is `complex<Promotion<T1, T2>::type>`, whichever applies first.

8.3.2 *Domain and range errors*

[math.fns.errors]

- 1 Behavior for domain, range, overflow, and underflow errors and for undefined values conforms to VSIP1.1 API, i.e., the result is undefined behavior. For functions on views defined using scalar functions, this behavior is extended element-wise, as defined in [math.definitions].

8.3.3 *Scalar functions*

[math.fns.scalar]

```
template <typename T>
T
acos(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `acos<scalar_f>`. An implementation is permitted to prevent instantiation of `acos<T>` for other choices of `T`.

Returns: The principal radian value in the range $[0, \pi]$ of the arccosine of `a`.

Note: Some specializations correspond to VSIPL function `vsip_acos_f`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
add(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `add<scalar_f, cscalar_f>` and `add<cscalar_f, cscalar_f>`. An implementation is permitted to prevent instantiation of `add<T1, T2>` for other choices of `T1` and `T2`.

Returns: The sum of the two operands.

Note: Some specializations correspond to VSIPL functions `vsip_rcadd_f` and `vsip_cadd_f`.

```
template <typename T>
T
arg(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `arg<scalar_f>`. An implementation is permitted to prevent instantiation of `arg<T>` for other choices of `T`.

Returns: The argument of `a` in radians. See `vsip_arg_f` for the mathematical specification.

Note: Some specializations correspond to VSIPL function `vsip_arg_f`.

```
template <typename T>
T
asin(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `asin<scalar_f>`. An implementation is permitted to prevent instantiation of `asin<T>` for other choices of `T`.

Returns: The principal radian value $[0, \pi]$ of the arcsine of `a`.

Note: Some specializations correspond to VSIPL function `vsip_asin_f`.

```
template <typename T>
T
atan(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `atan<scalar_f>`. An implementation is permitted to prevent instantiation of `atan<T>` for other choices of `T`.

Returns: The arctangent of `a`. See `vsip_atan_f` for the mathematical specification.

Note: Some specializations correspond to VSIPL function `vsip_atan_f`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
atan2(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `atan2<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `atan2<T1, T2>` for other choices of `T1` and `T2`.

Returns: The arctangent of the ratio of `a` and `b`. See `vsip_atan2_f` for the mathematical specification.

Note: Some specializations correspond to VSIPL function `vsip_atan2_f`.

```
template <typename T>
T
ceil(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `ceil<scalar_f>`. An implementation is permitted to prevent instantiation of `ceil<T>` for other choices of `T`.

Returns: The smallest integral value greater than or equal to the argument.

Note: Some specializations correspond to VSIPL function `vsip_ceil_f`. The return type is the same as the argument type, which is not necessarily an integral type.

```
template <typename T>
complex<T>
conj(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `conj<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `conj<T>` for other choices of `T` .

Returns: The conjugate of `a` .

Note: Some specializations correspond to VSIPL function `vsip_conj_`*f*`.`

```
template <typename T>
T
cos(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `cos<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `cos<T>` for other choices of `T` .

Returns: The cosine of `a` in radians.

Note: Some specializations correspond to VSIPL function `vsip_cos_`*f*`.`

```
template <typename T>
T
cosh(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `cosh<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `cosh<T>` for other choices of `T` .

Returns: The hyperbolic cosine of `a` .

Note: Some specializations correspond to VSIPL function `vsip_cosh_`*f*`.`

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
div(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `div<cscalar_`*f*`, scalar_`*f*`>`, `div<scalar_`*f*`, cscalar_`*f*`>`, and `div<cscalar_`*f*`, cscalar_`*f*`>` . An implementation is permitted to prevent instantiation of `div<T1, T2>` for other choices of `T1` and `T2` .

Returns: The quotient of the two operands.

Note: Some specializations correspond to VSIPL functions `vsip_crdiv_`*f* and `vsip_cdiv_`*f*`.`

```
template <typename T>
T
exp(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `exp<scalar_f>` and `exp<cscalar_f>`. An implementation is permitted to prevent instantiation of `exp<T>` for other choices of `T`.

Returns: The exponential of `a`, extended to the complex exponential if appropriate.

Note: Some specializations correspond to VSIPL functions `vsip_exp_f` and `vsip_cexp_f`. The exponential `exp(a)` of a complex `a` is $(\cos(\text{imag}(a)) + j \sin(\text{imag}(a))) \exp(\text{real}(a))$, where $j = \sqrt{-1}$.

```
template <typename T>
T
exp10(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `exp10<scalar_f>`. An implementation is permitted to prevent instantiation of `exp10<T>` for other choices of `T`.

Returns: The base-10 exponential of `a`, i.e., 10^a .

Note: Some specializations correspond to VSIPL function `vsip_exp10_f`.

```
template <typename T>
T
floor(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `floor<scalar_f>`. An implementation is permitted to prevent instantiation of `floor<T>` for other choices of `T`.

Returns: The largest integral value less than or equal to the argument.

Note: Some specializations correspond to VSIPL function `vsip_floor_f`. The return type is the same as the argument type, which is not necessarily an integral type.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
fmod(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: $b \neq 0.0$. The only specialization which must be supported is `fmod<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `fmod<T1, T2>` for other choices of `T1` and `T2`.

Returns: $a - \text{sign}(a) * \text{floor}(\text{mag}(a/b)) * \text{mag}(b)$, where $\text{sign}(a)$ is +1 if a is positive, 0 if a is zero, and -1 if a is negative.

Note: Some specializations correspond to VSIPL function `vsip_fmod_f`. The result of operating on signed values may differ from C++'s `%` operation on signed integral values.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
hypot(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `hypot<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `hypot<T1, T2>` for other choices of `T1` and `T2`.

Returns: $\text{sqrt}(a * a + b * b)$.

Note: Some specializations correspond to the VSIPL function `vsip_hypot_f`. Intermediate overflows will not occur.

```
template <typename T>
T
imag(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `imag<scalar_f>`. An implementation is permitted to prevent instantiation of `imag<T>` for other choices of `T`.

Returns: The imaginary portion of a .

Note: Some specializations correspond to VSIPL function `vsip_imag_f`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
jmul(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `jmul<cscalar_f, cscalar_f>`. An implementation is permitted to prevent instantiation of `jmul<T1, T2>` for other choices of `T1` and `T2`.

Returns: The product of a with the conjugate of b .

Note: Some specializations correspond to the scalar VSIPL function `vsip_cjmul_f`.

```
template <typename T>
T
log(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `log<scalar_f>` and `log<cscalar_f>`. An implementation is permitted to prevent instantiation of `log<T>` for other choices of T .

Returns: The natural logarithm of a , i.e., $\ln a$, extended to incorporate complex natural logarithms.

Note: Some specializations correspond to VSIPL functions `vsip_log_f` and `vsip_clof_f`. The complex logarithm $\ln a$ for complex a is $\ln|a| + j \arg(a)$, where $j = \sqrt{-1}$.

```
template <typename T>
T
log10(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `log10<scalar_f>`. An implementation is permitted to prevent instantiation of `log10<T>` for other choices of T .

Returns: The base-10 logarithm of a , i.e., $\log a$.

Note: Some specializations correspond to VSIPL function `vsip_log10_f`.

```
template <typename T>
T
mag(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `mag<scalar_f>`. An implementation is permitted to prevent instantiation of `mag<T>` for other choices of T .

Returns: The magnitude of a .

Note: Some specializations correspond to VSIPL function `vsip_cmag_f`.

```
template <typename T>
T
mag(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `mag<scalar_f>` and `mag<scalar_i>`. An implementation is permitted to prevent instantiation of `mag<T>` for other choices of `T`.

Returns: The absolute value of `a`.

Note: Some specializations correspond to VSIPL functions `vsip_mag_f` and `vsip_mag_i`.

```
template <typename T>
T
magsq(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `magsq<scalar_f>`. An implementation is permitted to prevent instantiation of `magsq<T>` for other choices of `T`.

Returns: The magnitude squared of `a`.

Note: Some specializations correspond to VSIPL function `vsip_cmagsq_f`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
max(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `max<scalar_f, scalar_f>` and `max<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `max<T1, T2>` for other choices of `T1` and `T2`.

Returns: Returns the maximum of `a` and `b`.

Note: Some specializations correspond to VSIPL functions `vsip_max_f` and `vsip_max_i`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
min(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `min<scalar_f, scalar_f>` and `min<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `min<T1, T2>` for other choices of `T1` and `T2`.

Returns: Returns the minimum of a and b .

Note: Some specializations correspond to VSIPL functions `vsip_min_f` and `vsip_min_i`.

```
template <typename T1,  
          typename T2>  
typename Promotion<T1, T2>::type  
mul(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `mul<cscalar_f, cscalar_f>` and `mul<scalar_f, cscalar_f>`. An implementation is permitted to prevent instantiation of `mul<T1, T2>` for other choices of $T1$ and $T2$.

Returns: The product of the two operands.

Note: Some specializations correspond to VSIPL functions `vsip_cmul_f` and `vsip_rcmul_f`.

```
template <typename T>  
T  
neg(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `neg<cscalar_f>`. An implementation is permitted to prevent instantiation of `neg<T>` for other choices of T .

Returns: $-a$.

Note: Some specializations correspond to the VSIPL function `vsip_cneg_f`.

```
template <typename T1,  
          typename T2>  
typename Promotion<T1, T2>::type  
pow(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `pow<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `pow<T1, T2>` for other choices of T .

Returns: The power function of a and b , i.e., a^b .

Note: Some specializations correspond to VSIPL function `vsip_pow_f`.

```
template <typename T>
T
real(complex<T> const& a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `real<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `real<T>` for other choices of `T` .

Returns: The real portion of `a` .

Note: Some specializations correspond to VSIP function `vsip_real_`*f*.

```
template <typename T>
T
recip(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `recip<cscalar_`*f*`>` . An implementation is permitted to prevent instantiation of `recip<T>` for other choices of `T` .

Returns: The reciprocal of the operand, extended to complex numbers as appropriate.

Note: Some specializations correspond to the VSIP function `vsip_crecip_`*f*.

```
template <typename T>
T
rsqrt(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `rsqrt<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `rsqrt<T>` for other choices of `T` .

Returns: The reciprocal of the square root of `a` .

Note: Some specializations correspond to the VSIP function `vsip_rsqrt_`*f*.

```
template <typename T>
T
sin(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `sin<scalar_`*f*`>` . An implementation is permitted to prevent instantiation of `sin<T>` for other choices of `T` .

Returns: The sine of `a` in radians.

Note: Some specializations correspond to VSIPPL function `vsip_sin_f`.

```
template <typename T>
T
sinh(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `sinh<scalar_f>`. An implementation is permitted to prevent instantiation of `sinh<T>` for other choices of `T`.

Returns: The hyperbolic sine of `a`.

Note: Some specializations correspond to VSIPPL function `vsip_sinh_f`.

```
template <typename T>
T
sqrt(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `sqrt<scalar_f>` and `sqrt<cscalar_f>`. An implementation is permitted to prevent instantiation of `sqrt<T>` for other choices of `T`.

Returns: The square root of `a`, extended to complex numbers as appropriate.

Note: Some specializations correspond to the VSIPPL functions `vsip_sqrt_f` and `vsip_csqr_f`. For a mathematical description of the complex square root, see the VSIPPL description of `vsip_csqr_f`.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
sub(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `sub<cscalar_f, scalar_f>`, `sub<scalar_f, cscalar_f>`, and `sub<cscalar_f, cscalar_f>`. An implementation is permitted to prevent instantiation of `sub<T1, T2>` for other choices of `T1` and `T2`.

Returns: `a - b`.

Note: Some specializations correspond to VSIPPL functions `vsip_csub_f`, `vsip_rsub_f`, and `vsip_crsub_f`.

```
template <typename T>
T
tan(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `tan<scalar_f>`. An implementation is permitted to prevent instantiation of `tan<T>` for other choices of `T`.

Returns: The tangent of `a` in radians.

Note: Some specializations correspond to VSIPL function `vsip_tan_f`.

```
template <typename T>
T
tan(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `tanh<scalar_f>`. An implementation is permitted to prevent instantiation of `tanh<T>` for other choices of `T`.

Returns: The hyperbolic tangent of `a`.

Note: Some specializations correspond to VSIPL function `vsip_tanh_f`.

8.3.4 Scalar functions used in element-wise extensions [math.fns.elements]

- 1 [Note: The following scalar functions are described here to ease specification of their element-wise extensions in [math.fns.elementwise] and [math.fns.scalarview] and the specification of the reduction functions in [math.fns.reductions] and [math.fns.reductidx].]
- 2 These functions need not be present in the `vsip` nor in any other namespace including the global namespace. Despite this, the functions are presented using the same format as in the rest of the document so that they can be referenced elsewhere.
- 3 If a VSIPL++ implementation does include these functions, they may be included in the `vsip` namespace.
- 4 [Note: Some functions appear both here and in [math.fns.scalar] because the set of required specializations differ. For these functions, the union of instantiation requirements should be used when interpreting element-wise and reduction requirements.]

```
template <typename T1,
         typename T2>
typename Promotion<T1, T2>::type
add(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `add<scalar_f, scalar_f>` and `add<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `add<T1, T2>` for other choices of `T1` and `T2`.

Returns: The sum of the two operands.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vadd_f`, `vsip_madd_f`, `vsip_svadd_f`, and `vsip_svadd_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2,
          typename T3>
typename Promotion<T1, typename Promotion<T2, T3>::type>::type
am(T1 a, T2 b, T3 c) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `am<cscalar_f, cscalar_f, cscalar_f>` and `am<scalar_f, scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `am<T1, T2, T3>` for other choices of `T1`, `T2`, and `T3`.

Returns: $(a + b) * c$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_cvam_f`, `vsip_vam_f`, `vsip_cvsam_f`, and `vsip_vsam_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
bool
land(bool a, bool b) VSIP_NOTHROW;
```

Returns: `a && b`.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vand_bl` and `vsip_mand_bl`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `and` is a keyword so it cannot be used as a function name. `land` abbreviates “logical and.”

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
band(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `band<scalar_i, scalar_i>`. Both T1 and T2 may not both be `bool`. An implementation is permitted to prevent instantiation of `band<T1, T2>` for other choices of T1 and T2.

Returns: `a & b`.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vand_i` and `vsip_mand_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `and` is a keyword so it cannot be used as a function name. `band` abbreviates “bitwise and.”

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
div(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `div<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `div<T1, T2>` for other choices of T1 and T2.

Returns: The quotient of the two operands.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vdiv_f`, `vsip_mdiv_f`, `vsip_svdiv_f`, and `vsip_vsdiv_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
bool
eq(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `eq<scalar_f, scalar_f>` and `eq<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `eq<T1, T2>` for other choices of T1 and T2.

Returns: `a == b`.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vleq_f` and `vsip_mleq_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions. Implementors may wish to implement `eq<cscalar_f,`

`cscalar_f`> and other complex arguments using the C++ library's == operator.

```
template <typename T>
complex<T>
euler(T x) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `euler<scalar_f>`. An implementation is permitted to prevent instantiation of `euler<T>` for other choices of `T`.

Returns: The complex number corresponding to the angle of a unit vector in the complex plane, i.e., $\exp(j * x)$ for argument `x`.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_veuler_f` and `vsip_meuler_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2,
          typename T3>
typename Promotion<T1, typename Promotion<T2, T3>:::type>::type
expoavg(T1 a, T2 b, T3 c) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `expoavg<scalar_f, cscalar_f, cscalar_f>` and `expoavg<scalar_f, scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `expoavg<T1, T2, T3>` for other choices of `T1`, `T2`, and `T3`.

Returns: $a * b + (1.0 - a) * c$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_cvexpoavg_f`, `vsip_cmexpoavg_f`, `vsip_vexpoavg_f`, and `vsip_mexpoavg_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
bool
ge(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `ge<scalar_f, scalar_f>` and `ge<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `ge<T1, T2>` for other choices of T1 and T2.

Returns: `a >= b`.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vlge_f`, `vsip_mlge_f`, `vsip_vlge_i`, and `vsip_mlge_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
bool
gt(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `gt<scalar_f, scalar_f>` and `gt<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `gt<T1, T2>` for other choices of T1 and T2.

Returns: `a > b`.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vlgt_f`, `vsip_mlgt_f`, `vsip_vlgt_i`, and `vsip_mlgt_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
bool
le(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `le<scalar_f, scalar_f>` and `le<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `le<T1, T2>` for other choices of T1 and T2.

Returns: `a <= b`.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vlle_f`, `vsip_mlle_f`, `vsip_vlle_i`, and `vsip_mlle_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2>
bool
lt(T1 a, T2 b) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are `lt<scalar_f, scalar_f>` and `lt<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `lt<T1, T2>` for other choices of T1 and T2.

Returns: $a < b$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vllt_f`, and `vsip_mllt_f`, `vsip_vllt_i`, and `vsip_mllt_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2,
          typename T3>
typename Promotion<T1, typename Promotion<T2, T3>::type>::type
ma(
    T1 a,
    T2 b,
    T3 c)
VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are `ma<cscalar_f, cscalar_f, cscalar_f>` and `ma<scalar_f, scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `ma<T1, T2, T3>` for other choices of T1, T2, and T3.

Returns: $(a * b) + c$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_cvma_f`, `vsip_vma_f`, `vsip_cvmsa_f`, `vsip_vmsa_f`, `vsip_cvmsma_f`, `vsip_vmsma_f`, `vsip_cvmsa_f`, and `vsip_vmsa_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
maxmg(T1 a, T2 b) VSIP_NOTHROW;

```

Requires: The only specialization which must be supported is `maxmg<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `maxmg<T1, T2>` for other choices of `T1` and `T2`.

Returns: $\max(|a|, |b|)$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vmaxmg_f` and `vsip_mmaxmg_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
maxmgsq(complex<T1> const& a, complex<T2> const& b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `maxmgsq<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `maxmgsq<T1, T2>` for other choices of `T1` and `T2`.

Returns: $\max(|a|^2, |b|^2)$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vcmaxmgsq_f` and `vsip_mcmxmgsg_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
minmg(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `minmg<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `minmg<T1, T2>` for other choices of `T1` and `T2`.

Returns: $\min(|a|, |b|)$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vminmg_f` and `vsip_mminmg_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
minmgsq(complex<T1> const&, complex<T2> const&) VSIP_NOTHROW;

```

Requires: The only specialization which must be supported is `minmgsq<scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `minmgsq<T1, T2>` for other choices of T1 and T2.

Returns: $\min(|a|^2, |b|^2)$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vcmimnmsq_f` and `vsip_mcmimnmsq_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2,
          typename T3>
typename Promotion<T1, typename Promotion<T2, T3>::type>::type
msb(
    T1 a,
    T2 b,
    T3 c)
VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are `msb<cscalar_f, cscalar_f, cscalar_f>` and `msb<scalar_f, scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `msb<T1, T2, T3>` for other choices of T1, T2, and T3.

Returns: $(a * b) - c$.

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_cvmsb_f` and `vsip_vmsb_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```

template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
mul(T1 a, T2 b) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are `mul<scalar_f, scalar_f>` and `mul<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `mul<T1, T2>` for other choices of `T1` and `T2`.

Returns: The product of the two operands.

Note: Some specializations correspond to the scalar versions of the VSIPL functions `vsip_vmul_f`, `vsip_mmul_f`, `vsip_vmul_i`, `vsip_mmul_i`, and `vsip_svmul_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
bool
ne(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `ne<scalar_f, scalar_f>` and `ne<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `ne<T1, T2>` for other choices of `T1` and `T2`.

Returns: `a != b`.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vlne_f`, `vsip_mlne_f`, `vsip_vlne_i`, and `vsip_mlne_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T>
T
neg(T a) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `neg<scalar_f>` and `neg<scalar_i>`. An implementation is permitted to prevent instantiation of `neg<T>` for other choices of `T`.

Returns: `-a`.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vneg_f`, `vsip_mneg_f`, `vsip_vneg_i` and `vsip_mneg_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
bool
lnot(bool a) VSIP_NOTHROW;
```

Returns: !a .

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vnot_bl` and `vsip_mnot_bl`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `not` is a keyword so it cannot be used as a function name. `lnot` abbreviates “logical not.”

```
template <typename T>
T
bnot (T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `bnot<scalar_i>` . `T` may not be `bool` . An implementation is permitted to prevent instantiation of `bnot<T>` for other choices of `T` .

Returns: ~a .

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vnot_i` and `vsip_mnot_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `not` is a keyword so it cannot be used as a function name. `bnot` abbreviates “bitwise not.”

```
bool
lor (bool a, bool b) VSIP_NOTHROW;
```

Returns: a || b .

Note: Some specializations correspond to the scalar versions of VSIP functions `vsip_vor_bl` and `vsip_mor_bl`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `or` is a keyword so it cannot be used as a function name. `lor` abbreviates “logical or.”

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
bor (T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `bor<scalar_i, scalar_i>`. Both `T1` and `T2` may not be `bool`. An implementation is permitted to prevent instantiation of `bor<T1, T2>` for other choices of `T1` and `T2`.

Returns: `a | b`.

Note: Some specializations correspond to the scalar versions of VSIPPL functions `vsip_vor_i` and `vsip_mor_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `or` is a keyword so it cannot be used as a function name. `bor` abbreviates “bitwise or.”

```
template <typename T>
T
recip(T a) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `recip<scalar_f>`. An implementation is permitted to prevent instantiation of `recip<T>` for other choices of `T`.

Returns: The reciprocal of the operand, extended to complex numbers as appropriate.

Note: Some specializations correspond to the scalar VSIPPL functions `vsip_vrecip_f` and `vsip_mrecip_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2,
          typename T3>
typename Promotion<T1, typename Promotion<T2, T3>::type>::type
sbm(T1 a, T2 b, T3 c) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `sbm<cscalar_f, cscalar_f, cscalar_f>` and `sbm<scalar_f, scalar_f, scalar_f>`. An implementation is permitted to prevent instantiation of `sbm<T1, T2, T3>` for other choices of `T1`, `T2`, and `T3`.

Returns: `(a - b) * c`.

Note: Some specializations correspond to the scalar versions of VSIPPL functions `vsip_cvsbm_f` and `vsip_vsbm_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T>
T
sq(T) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `sq<scalar_f>`. An implementation is permitted to prevent instantiation of `sq<T>` for other choices of `T`.

Returns: The square of the operand, i.e., product with itself.

Note: Some specializations correspond to the scalar VSIPL functions `vsip_vsqr_f` and `vsip_msqr_f`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
sub(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `sub<scalar_f, scalar_f>` and `sub<scalar_i, scalar_i>`. An implementation is permitted to prevent instantiation of `sub<T1, T2>` for other choices of `T1` and `T2`.

Returns: `a - b`.

Note: Some specializations correspond to the scalar versions of the VSIPL functions `vsip_vsub_f`, `vsip_msub_f`, `vsip_vsub_i`, `vsip_msub_i`, `vsip_svsub_f`, and `vsip_svsub_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

```
bool
lxor(bool, bool) VSIP_NOTHROW;
```

Returns: `((a && !b) || (!a && b))`.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vxor_bl` and `vsip_mxor_bl`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `xor` is a keyword so it cannot be used as a function name. `lxor` abbreviates “logical xor.”

```
template <typename T1,
          typename T2>
typename Promotion<T1, T2>::type
bxor(T1, T2) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported is `bxor<scalar_i, scalar_i>`. Both `T1` and `T2` cannot both be `bool`. An implementation is permitted to prevent instantiation of `bxor<T1, T2>` for other choices of `T1` and `T2`.

Returns: $a \wedge b$ where `bool` values are interpreted as either 0 for false or 1 for true.

Note: Some specializations correspond to the scalar versions of VSIPL functions `vsip_vxor_i` and `vsip_mxor_i`. This scalar function need not be defined, but this function may be used to specify element-wise extensions or reduction functions.

In C++, `xor` is a keyword so it cannot be used as a function name. `bxor` abbreviates “bitwise xor.”

8.3.5 Scalar functions and their element-wise extensions [math.fns.elementwise]

1 These function specifications have two portions:

- a scalar version defined in terms of a VSIPL function or a scalar function in [math.fns.scalar] or [math.fns.elements] and
- an element-wise extension to views.

2 Unless otherwise noted, the element-wise extension of scalar unary function `f` to a constant view `const_View`

Requires: One parameter `const_View v` where `const_View` is a constant view class such that `f(val)` is valid C++ where `val` has the type `View::value_type`.

Returns: A const view with type `const_View` and value type the same as the type of `f(const_View::value_type)` and having a domain element-conformant to `v`'s domain. Each value equals `f(val)` where `val` is the corresponding value in `v`.

Note: The type of the block of the returned view may be different than the type of `v`'s block.

3 The element-wise extension of a scalar binary function f to a constant view `const_View` similarly

Requires: Two parameters `const_View1 v` and `const_View2 w` where `const_View1` and `const_View2` are view classes such that $f(v_val, w_val)$ is valid C++ where `v_val` and `w_val` have types `const_View1::value_type` and `const_View2::value_type`, respectively. `v` and `w` are element-conformant.

Returns: A constant view with type `const_View` and value type the same as the type of $f(const_View1::value_type, const_View2::value_type)$ and having a domain element-conformant to `v`'s domain. Each value equals $f(val1, val2)$ where `val1` and `val2` are the corresponding values in `v` and `w`, respectively.

Note: The type of the block of the returned view may be different than the types of `v`'s and `w`'s blocks.

4 Element-wise extension of n -ary functions to a view is analogously specified.

5 The scalar unary, binary, and ternary functions listed in Table 8.1 all conform to the above paragraphs by having element-wise extensions. The table's third column indicates which view classes must support element-wise extensions using the following abbreviations:

V: Vector

M: Matrix

T: Tensor

An implementation is permitted to prevent instantiation for other choices of `View`, `View1`, and `View2`.

Table 8.1: Integral, real, complex, and boolean functions with element-wise extensions

function	meaning	views
<code>acos</code>	arccosine	VM
<code>add</code>	addition	VM
<code>am</code>	addition then product	V
<code>land</code>	logical and	VM
<code>band</code>	bitwise and	VM
<code>arg</code>	real scalar in radians	VM
<code>asin</code>	arcsine	VM
<code>atan</code>	arctangent	VM
<code>atan2</code>	two-argument arctangent	VM

function	meaning	views
ceil	ceiling	VM
conj	complex conjugate	VM
cos	cosine	VM
cosh	hyperbolic cosine	VM
div	division	VM
eq	equality	VM
euler	complex exponential	VM
exp	exponential	VM
exp10	base-10 exponential	VM
floor	floor	VM
fmod	remainder	VM
ge	greater than or equal	VM
gt	greater than	VM
hypot	square root of sum of squares	VM
imag	imaginary part of complex	VM
jmul	product with complex conjugate	VM
le	less than or equal	VM
log	natural logarithm	VM
log10	base-10 logarithm	VM
lt	less than	VM
ma	product and addition	V
mag	absolute value	VM
magsq	squared magnitude	VM
max	maximum	VM
maxmg	maximum magnitude	VM
maxmgsq	maximum magnitude squared	VM
min	minimum	VM
minmg	minimum magnitude	VM
minmgsq	minimum magnitude squared	VM
msb	product and subtraction	V
mul	product	VM
ne	not equal	VM
neg	negation	VM
lnot	logical not	VM
bnot	bitwise not	VM
lor	logical or	VM
bor	bitwise or	VM
pow	power function	VM
real	real part of complex	VM
recip	reciprocal	VM
rsqrt	reciprocal square root	VM
sbm	subtraction and product	V
sin	sine	VM
sinh	hyperbolic sine	VM

function	meaning	views
sq	square	VM
sqrt	square root	VM
sub	subtraction	VM
tan	tangent	VM
tanh	hyperbolic tangent	VM
lxor	logical exclusive-or	VM
bxor	bitwise exclusive-or	VM

```

template <typename T1,
         typename T2,
         template <typename, typename> class const_View,
         typename Block1,
         typename Block2>
const_View<complex<typename Promotion<T1, T2>::type>, un-
specified>
cmplx (
    const_View<T1, Block1> v,
    const_View<T2, Block2> w) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported have T1 and T2 both equal to `scalar_f` and `const_View` equal to `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation with other values. `v` and `w` must be element conformant.

Returns: The element-wise extension of the `complex(T1, T2)` constructor to views.

Note: Some specializations correspond to the VSIPL functions `vsip_vcplx_f` and `vsip_mcplx_f`.

8.3.6 *Element-wise functions with scalar and view operands* [**math.fns.-scalarview**]

- 1 These function specifications define an element-wise extension of a function on a view and a scalar or on a scalar and a view using a scalar function defined in terms of a VSIPL binary function or a scalar binary function in [`math.fns.scalar`] or [`math.fns.elements`].
- 2 The element-wise extension of a scalar binary function `f` to a view and a scalar

Requires: Two parameters `const_View1 v` and `scalartype s` where `const_View1` is a constant view class such that `f(val, s)` is valid

C++ where `val` has type `View1::value_type` and `s` has type `scalartype`.

Returns: A constant view with value type the same as the type of `f(const_View1::value_type, scalartype)` and having a domain element-conformant to `v`'s domain. Each value equals `f(val, s)` where `val` is the corresponding value in `v`.

Note: The type of the block of the returned view may be different than the type of `v`'s block.

3 The element-wise extension of a scalar binary function `f` to a scalar and a view is specified analogously.

4 The element-wise extension of a scalar ternary function `f` to a scalar, a view, and a view

Requires: Three parameters `scalartype s`, `const_View1 v`, and `const_View2 w` where `const_View1` and `const_View2` are view classes such that `f(s, v_val, w_val)` is valid C++, `s` has type `scalartype`, `v_val` has type `const_View1::value_type`, `w_val` has type `const_View2::value_type`. `v` and `w` are element-conformant.

Returns: A view with value type the same as the type of `f(scalartype, const_View1::value_type, const_View2::value_type)` and having a domain element-conformant to `v`'s domain. Each value equals `f(s, v_val, w_val)` where `v_val` and `w_val` are corresponding values in `v` and `w`.

Note: The type of the block of the returned view may be different than the types of `v`'s and `w`'s blocks.

5 The element-wise extension of a scalar ternary function `f` to two views and a scalar is specified analogously. The element-wise extension of a scalar ternary function `f` to a view, a scalar, and a view is specified analogously. The element-wise extension of a scalar ternary function `f` to a view and two scalars is specified analogously.

6 The functions listed in Table 8.2 all conform to the above paragraphs. The right-most column indicates which classes must support element-wise extensions using the following abbreviations:

V: `const_Vector`

M: `const_Matrix`

T: `const_Tensor`

`const_View::value_type`, `const_View1::value_type`, `const_View2::value_type`, `T`, `T1`, and `T2` can all represent distinct types. `const_View`, `const_View1`, and `const_View2` each abbreviate `const_View<Tp, Blockp>` for appropriate values of `Tp` and `Blockp`.

Table 8.2: Functions with elementwise extensions on views and scalars

function	meaning	views
<code>add(T, const_View)</code>	addition	VM
<code>add(const_View, T)</code>	addition	VM
<code>am(const_View1, T, const_View2)</code>	addition and product	V
<code>div(T, const_View)</code>	division	VM
<code>div(const_View, T)</code>	division	VM
<code>expoavg(T, const_View1, const_View2)</code>	exponential average	VM
<code>ma(const_View1, T1, T2)</code>	multiply and add	V
<code>ma(const_View1, T, const_View2)</code>	multiply and add	V
<code>ma(const_View1, const_View2, T)</code>	multiply and add	V
<code>mul(T, const_View)</code>	product	VM
<code>mul(const_View, T)</code>	product	VM
<code>sub(T, const_View)</code>	subtraction	VM
<code>sub(const_View, T)</code>	subtraction	VM

7 Instantiation requirements follow directly from the instantiation requirements for the underlying scalar function. [*Example:* The only specializations of the element-wise extension of `am` to a view, a scalar, and another view which must be supported have `const_View1::value_type`, `T`, and `const_View2::value_type` all equal and all equal to `cscalar_f` or `scalar_f` because the only specializations of the scalar function which must be supported are `am<cscalar_f, cscalar_f, cscalar_f>` and `am<scalar_f, scalar_f, scalar_f>`.]

8 An implementation is permitted to prevent instantiation of `div(const_View, T)` for complex `T`.

8.3.7 Element-wise extension of user-specified functions [math.fns.userelt]

1 Element-wise extensions of user-specified functions element-wise apply a function pointer or function object to the appropriate number of views.

- 2 [Note: VSIPL++ specifies binary functions that correspond to VSIPL `vsip_sbinary` and `vsip_sbool_p` functions. `bool` is a C++ keyword so using it as a function name is forbidden by C++.]
- 3 [Note: The VSIPL++ unary, binary, and ternary functions each provide portions of the VSIPL `vsip_snary` functionality.]
- 4 [Note: unary and binary are overloaded to accept different types of function objects and pointers. The first version takes a function object and requires the user to explicitly specify the return type as a template argument. The second version takes a function pointer and also requires an explicit return type template argument. The third version accepts function objects obeying (ISO14882, [lib.function.objects]). These function objects use internal type definitions to indicate the argument and return types so no explicit return type template argument is required. The `<functional>` header file must be included to use this. ternary has only two versions because C++ does not specify a function object with three arguments.]

```
template <typename OutputType,
          typename UnaryFunction,
          template <typename, typename> class const_View,
          typename InputType,
          typename Block>
const_View<OutputType, unspecified>
unary (
    UnaryFunction          f,
    const_View<InputType, Block> v);
```

Requires: If `input` is an instance of `InputType`, the expression `f(input)` must yield a value with type `OutputType`. The only specializations which must be supported must satisfy one of these groups of constraints:

- `UnaryFunction` is a function pointer. `InputType` and `OutputType` must have the same type. If `const_View` is `const_Vector`, then `InputType` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `InputType` must equal `scalar_f` or `scalar_i`.
- `UnaryFunction` is a function pointer. `InputType` equals `index_type`. `const_View` equals `const_Vector`. `OutputType` equals `scalar_f`, `scalar_i`, `bool`, `Index<1>`, `Index<2>`, or `Index<3>`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType` or `OutputType`.

Returns: A constant view element-conformant with `v` where each value equals `f(val)` where `val` is the corresponding value in `v`.

Note: The order that `f` is applied to the values in `v` is unspecified. The type of the block of the returned view may be different than the type of `v`'s block. This function corresponds to VSIPL functions `vsip_vunary_f`, `vsip_munary_f`, `vsip_tunary_f`, `vsip_vunary_i`, `vsip_munary_i`, `vsip_tunary_i`, `vsip_vunary_vi`, `vsip_vunary_mi`, and `vsip_vunary_ti`. It also corresponds to `vsip_vnary_f`, `vsip_vnary_i`, `vsip_vnary_bl`, `vsip_vnary_vi`, `vsip_vnary_mi`, and `vsip_vnary_ti`.

```
template <typename                               OutputType,
         template <typename, typename> class const_View,
         typename                               InputType,
         typename                               Block>
const_View<OutputType, unspecified>
unary(
    OutputType (*f)(InputType),
    const_View<InputType, Block> v);
```

Requires: The only specializations which must be supported must satisfy one of these groups of constraints:

- `InputType` and `OutputType` must have the same type. If `const_View` is `const_Vector`, then `InputType` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `InputType` must equal `scalar_f` or `scalar_i`.
- `InputType` equals `index_type`. `const_View` equals `const_Vector`. `OutputType` equals `scalar_f`, `scalar_i`, `bool`, `Index<1>`, `Index<2>`, or `Index<3>`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType` or `OutputType`.

Returns: A view element-conformant with `v` where each value equals `f(val)` where `val` is the corresponding value in `v`.

Note: This function overloads the more general unary function to provide explicit support for function pointers. The order that `f` is applied to the values in `v` is unspecified. The type of the block of the returned view may

be different than the type of v 's block. This function corresponds to VSIPL functions `vsip_vunary_f`, `vsip_munary_f`, `vsip_tunary_f`, `vsip_vunary_i`, `vsip_munary_i`, `vsip_tunary_i`, `vsip_vunary_vi`, `vsip_vunary_mi`, and `vsip_vunary_ti`. It also corresponds to `vsip_vnary_f`, `vsip_vnary_i`, `vsip_vnary_bl`, `vsip_vnary_vi`, `vsip_vnary_mi`, and `vsip_vnary_ti`.

```
template <typename UnaryFunction,
         template <typename, typename> class const_View,
         typename Block>
const_View<typename UnaryFunction::result_type, unspecified>
unary(
    UnaryFunction f,
    const_View<typename UnaryFunction::argument_type, Block>);
```

Requires: `UnaryFunction` must be a class such that `UnaryFunction::result_type` and `UnaryFunction::argument_type` are defined and have a function operator `typename UnaryFunction::result_type operator() (typename UnaryFunction::argument_type)`. The only specializations which must be supported must satisfy one of these groups of constraints:

- `UnaryFunction::argument_type` and `UnaryFunction::result_type` must have the same type. If `const_View` is `const_Vector`, then `UnaryFunction::argument_type` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `UnaryFunction::argument_type` must equal `scalar_f` or `scalar_i`.
- `UnaryFunction::argument_type` equals `index_type`. `const_View` equals `const_Vector`. `UnaryFunction::result_type` equals `scalar_f`, `scalar_i`, `bool`, `Index<1>`, `Index<2>`, or `Index<3>`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `UnaryFunction::argument_type` or `UnaryFunction::result_type`.

Returns: A constant view element-conformant with v where each value equals $f(val)$ where val is the corresponding value in v .

Note: This function overloads the more general unary function to provide explicit support for function objects obeying (ISO14882, [lib.function.objects]). The order that f is applied to the values in v is unspecified. The type of the block of the returned view may be different than the type of v 's block. This function corresponds to VSIPL functions `vsip_vunary_f`, `vsip_munary_f`, `vsip_tunary_f`, `vsip_vunary_i`, `vsip_munary_i`, `vsip_tunary_i`, `vsip_vunary_vi`, `vsip_vunary_mi`, and `vsip_vunary_ti`. It also corresponds to `vsip_vnary_f`, `vsip_vnary_i`, `vsip_vnary_bl`, `vsip_vnary_vi`, `vsip_vnary_mi`, and `vsip_vnary_ti`.

```
template <typename                               OutputType,
          typename                               BinaryFunction,
          template <typename, typename> class   const_View,
          typename                               InputType0,
          typename                               Block0,
          typename                               InputType1,
          typename                               Block1>
const_View<OutputType, unspecified>
binary(
    BinaryFunction                               f,
    const_View<InputType0, Block0> const v,
    const_View<InputType1, Block1>          w);
```

Requires: If `input0` is an instance of `InputType0` and `input1` is an instance of `InputType1`, the expression `f(input0, input1)` must yield a value with type `OutputType`. `v` and `w` must be element-conformant. The only specializations which must be supported must satisfy one of these groups of constraints:

- `BinaryFunction` is a function pointer. `InputType0` and `OutputType` must have the same type. `InputType0` equals `InputType1`. If `const_View` is `const_Vector`, then `InputType0` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `InputType0` must equal `scalar_f`, `scalar_i`, or `bool`.
- `BinaryFunction` is a function pointer. `OutputType` must equal `bool`. `InputType0` equals `InputType1`. If `const_View` is `const_Vector`, then `InputType0` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If

`const_View` is `const_Matrix` or `const_Tensor`, then `InputType0` must equal `scalar_f` or `scalar_i`.

- `BinaryFunction` is a function pointer. `InputType0` equals `InputType1`. `InputType0` equals `index_type`. `const_View` equals `const_Matrix`. `OutputType` equals `scalar_f`, `scalar_i`, or `bool`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType0`, `InputType1`, or `OutputType`.

Returns: A constant view element-conformant with `v` where each value equals `f(val0, val1)` where `val0` and `val1` are the corresponding values in `v` and `w`, respectively.

Note: The order that `f` is applied to the values in `v` and `w` is unspecified. The type of the block of the returned view may be different than the types of `v`'s and `w`'s blocks. This function corresponds to VSIPL functions `vsip_vbinary_f`, `vsip_mbinary_f`, `vsip_tbinary_f`, `vsip_vbinary_i`, `vsip_mbinary_i`, `vsip_tbinary_i`, `vsip_vbinary_vi`, `vsip_vbinary_mi`, `vsip_vbinary_ti`, and `vsip_mbinary_bl`, `vsip_tbinary_bl`. It also corresponds to `vsip_vbool_f`, `vsip_mbool_f`, `vsip_tbool_f`, `vsip_vbool_i`, `vsip_mbool_i`, `vsip_tbool_i`, `vsip_vbool_vi`, `vsip_vboolmi`, and `vsip_vboolti`. It also corresponds to `vsip_mnary_f`, `vsip_mnary_i`, and `vsip_mnary_bl`.

```
template <typename                               OutputType,
         template <typename, typename> class const_View,
         typename                               InputType0,
         typename                               Block0,
         typename                               InputType1,
         typename                               Block1>
const_View<OutputType, unspecified>
binary(
    OutputType (*f)( InputType0, InputType1),
    const_View<InputType0, Block0> v,
    const_View<InputType1, Block1> w);
```

Requires: If `input0` is an instance of `InputType0` and `input1` is an instance of `InputType1`, the expression `f(input0, input1)` must yield a value with type `OutputType`. `v` and `w` must be element-conformant. The only specializations which must be supported must satisfy one of these groups of constraints:

- `InputType0` and `OutputType` must have the same type. `InputType0` equals `InputType1`. If `const_View` is `const_Vector`, then `InputType0` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `InputType0` must equal `scalar_f`, `scalar_i`, or `bool`.
- `OutputType` must equal `bool`. `InputType0` equals `InputType1`. If `const_View` is `const_Vector`, then `InputType0` must equal `scalar_f`, `scalar_i`, `Index<1>`, `Index<2>`, or `Index<3>`. If `const_View` is `const_Matrix` or `const_Tensor`, then `InputType0` must equal `scalar_f` or `scalar_i`.
- `InputType0` equals `InputType1`. `InputType0` equals `index_type`. `const_View` equals `const_Matrix`. `OutputType` equals `scalar_f`, `scalar_i`, or `bool`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType0`, `InputType1`, or `OutputType`.

Returns: A constant view element-conformant with `v` where each value equals `f(val0, val1)` where `val0` and `val1` are the corresponding values in `v` and `w`, respectively.

Note: This function overloads the more general binary function to provide explicit support for function pointers. The order that `f` is applied to the values in `v` and `w` is unspecified. The type of the block of the returned view may be different than the types of `v`'s and `w`'s blocks. This function corresponds to VSIPPL functions `vsip_vbinary_f`, `vsip_mbinary_f`, `vsip_tbinary_f`, `vsip_vbinary_i`, `vsip_mbinary_i`, `vsip_tbinary_i`, `vsip_vbinary_vi`, `vsip_vbinary_mi`, `vsip_vbinary_ti`, and `vsip_mbinary_bl`, `vsip_tbinary_bl`. It also corresponds to `vsip_vbool_f`, `vsip_mbool_f`, `vsip_tbool_f`, `vsip_vbool_i`, `vsip_mbool_i`, `vsip_tbool_i`, `vsip_vbool_vi`, `vsip_vbool_mi`, and `vsip_vbool_ti`. It also corresponds to `vsip_mnary_f`, `vsip_mnary_i`, and `vsip_mnary_bl`.

```
template <typename                               BinaryFunction,
         template <typename, typename> class const_View,
         typename                               Block0,
         typename                               Block1>
const_View<typename BinaryFunction::result_type, unspecified>
```

binary(

```
BinaryFunction f,
const_View<typename BinaryFunction::first_argument_type, Block0>,
const_View<typename BinaryFunction::second_argument_type, Block1>);
```

Requires: BinaryFunction must be a class such that BinaryFunction::result_type, BinaryFunction::first_argument_type, and BinaryFunction::second_argument_type are defined and have a function operator typename BinaryFunction::result_type operator() (typename BinaryFunction::first_argument_type, typename BinaryFunction::second_argument_type). v and w must be element-conformant. The only specializations which must be supported must satisfy one of these groups of constraints:

- BinaryFunction::first_argument_type and BinaryFunction::result_type must have the same type. BinaryFunction::first_argument_type equals BinaryFunction::second_argument_type . If const_View is const_Vector, then BinaryFunction::first_argument_type must equal scalar_f, scalar_i, Index<1>, Index<2>, or Index<3> . If const_View is const_Matrix or const_Tensor, then BinaryFunction::first_argument_type must equal scalar_f, scalar_i, or bool .
- BinaryFunction::result_type must equal bool . BinaryFunction::first_argument_type equals BinaryFunction::second_argument_type . If const_View is const_Vector, then BinaryFunction::first_argument_type must equal scalar_f, scalar_i, Index<1>, Index<2>, or Index<3> . If const_View is const_Matrix or const_Tensor, then BinaryFunction::first_argument_type must equal scalar_f or scalar_i .
- BinaryFunction::first_argument_type equals BinaryFunction::second_argument_type . BinaryFunction::first_argument_type equals index_type . const_View equals const_Matrix . BinaryFunction::result_type equals scalar_f, scalar_i, or bool .

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `BinaryFunction::first_argument_type`, `BinaryFunction::second_argument_type`, or `BinaryFunction::result_type`.

Returns: A constant view element-conformant with `v` where each value equals `f(val0, val1)` where `val0` and `val1` are the corresponding values in `v` and `w`, respectively.

Note: This function overloads the more general binary function to provide explicit support for function objects obeying (ISO14882, [lib.function.objects]). The order that `f` is applied to the values in `v` and `w` is unspecified. The type of the block of the returned view may be different than the types of `v`'s and `w`'s blocks. This function corresponds to VSIPL functions `vsip_vbinary_f`, `vsip_mbinary_f`, `vsip_tbinary_f`, `vsip_vbinary_i`, `vsip_mbinary_i`, `vsip_tbinary_i`, `vsip_vbinary_vi`, `vsip_vbinary_mi`, `vsip_vbinary_ti`, and `vsip_mbinary_bl`, `vsip_tbinary_bl`. It also corresponds to `vsip_vbool_f`, `vsip_mbool_f`, `vsip_tbool_f`, `vsip_vbool_i`, `vsip_mbool_i`, `vsip_tbool_i`, `vsip_vbool_vi`, `vsip_vbool_mi`, and `vsip_vbool_ti`. It also corresponds to `vsip_mnary_f`, `vsip_mnary_i`, and `vsip_mnary_bl`.

```
template <typename                               OutputType,
         typename                               TernaryFunction,
         template <typename, typename> class   const_View,
         typename                               InputType0,
         typename                               Block0,
         typename                               InputType1,
         typename                               Block1,
         typename                               InputType2,
         typename                               Block2>
const_View<OutputType, unspecified>
ternary (
    TernaryFunction                               f,
    const_View<InputType0, Block0> v,
    const_View<InputType1, Block1> w,
    const_View<InputType2, Block2> x);
```

Requires: If `input0` is an instance of `InputType0`, `input1` is an instance of `InputType1`, and `input2` is an instance of `InputType2`, the ex-

pression `f(input0, input1, input2)` must yield a value with type `OutputType`. `v`, `w`, and `x` must be element-conformant. The only specializations which must be supported must satisfy this group of constraints:

- TernaryFunction is a function pointer. `InputType0` equals `InputType1` and `InputType2`. `InputType0` equals `index_type`. `const_View` equals `const_Matrix`. `OutputType` equals `scalar_f`, `scalar_i`, or `bool`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType0`, `InputType1`, `InputType2`, or `OutputType`.

Returns: A constant view element-conformant with `v` where each value equals `f(val0, val1, val2)` where `val0`, `val1`, and `val2` are the corresponding values in `v`, `w`, and `x`, respectively.

Note: The order that `f` is applied to the values in `v`, `w`, and `x` is unspecified. The type of the block of the returned view may be different than the types of `v`'s, `w`'s, and `x`'s blocks. This function corresponds to VSIPPL functions `vsip_ternary_f`, `vsip_ternary_i`, and `vsip_ternary_bl`.

```
template <typename                               OutputType,
          template <typename, typename> class  const_View,
          typename                               InputType0,
          typename                               Block0,
          typename                               InputType1,
          typename                               Block1,
          typename                               InputType2,
          typename                               Block2>
const_View<OutputType, unspecified>
ternary(
    OutputType (*f)(InputType0, InputType1, InputType2),
    const_View<InputType0, Block0> v,
    const_View<InputType1, Block1> w,
    const_View<InputType2, Block2> x);
```

Requires: `v`, `w`, and `x` must be element-conformant. The only specializations which must be supported must satisfy this group of constraints:

- `InputType0` equals `InputType1` and `InputType2`. `InputType0` equals `index_type`. `const_View` equals `const_Matrix`. `OutputType` equals `scalar_f`, `scalar_i`, or `bool`.

Regardless of constraint group, an implementation is permitted to prevent instantiation for complex `InputType0`, `InputType1`, `InputType2`, or `OutputType`.

Returns: A constant view element-conformant with `v` where each value equals `f(val0, val1, val2)` where `val0`, `val1`, and `val2` are the corresponding values in `v`, `w`, and `x`, respectively.

Note: This function overloads the more general `ternary` function to provide explicit support for function pointers. The order that `f` is applied to the values in `v`, `w`, and `x` is unspecified. The type of the block of the returned view may be different than the types of `v`'s, `w`'s, and `x`'s blocks. This function corresponds to VSIPPL functions `vsip_ternary_f`, `vsip_ternary_i`, and `vsip_ternary_bl`.

8.3.8 Reduction functions

[**math.fns.reductions**]

- 1 A *reduction function*, given a constant view, returns a scalar computed using all the view's values.
- 2 An *accumulation operator* `F` with *base value* `bv` applied to a constant view `v` yields answer `ans` determined by this algorithm: First, `ans = bv`. For each value `val` in `v`, `ans = F(ans, val)`. The order for accessing the values in `v` is unspecified.
- 3 An *accumulation operator* `F` with *base value* `bv` applied to a set `S` of values yields answer `ans` determined by this algorithm: First, `ans = bv`. For each `s` in `S`, `ans = F(ans, s)`. The order for traversing `S` is unspecified.
- 4 [*Note:* Most reduction functions `f` from a constant view `const_View v` to a scalar can be defined using three pieces:
 - a function `s` on the view's domain's size returning a scalar with type `t`,
 - an accumulation operator `F` taking a scalar with type `t` and a `v` value, `x` returning a scalar with type `t`, and
 - a function `c` taking a `const_View::value_type` and yielding scalars with type `t`.

$$f(v) = s(v.size()) * F_{i \in v's \text{ domain}} c(v(i)).]$$

- 5 [*Example:* The mean of a constant view can be represented using

- $s(sz) = 1/sz,$

- F is the addition operator, and
- $c(val) = val$.

For example, the mean of a Matrix m with a $M \times N$ domain is $1/(MN) \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} m.get(i, j)$.]

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
alltrue(
    const_View<T, Block> v)
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as `bool` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Returns: If T is `bool`, the result of the application of accumulation operator `land` with base value `!(T())` applied to v . Otherwise, the result of the application of accumulation operator `band` with base value `~(T())` applied to v .

Note: Some specializations correspond to the VSIPL functions `vsip_valltrue_bl` and `vsip_malltrue_bl`.

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
anytrue(const_View<T, Block> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as `bool` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Returns: If T is `bool`, the result of the application of accumulation operator `lor` with base value `T()` applied to v . Otherwise, the result of the application of accumulation operator `bor` with base value `T()` applied to v .

Note: Some specializations correspond to the VSIPL functions `vsip_vanytrue_bl` and `vsip_manytrue_bl`.

```
template <typename                                T,
         template <typename, typename> class const_View,
         typename                                Block>
T
meanval(const_View<T, Block> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as *scalar_f* or *cscalar_f* and *const_View* the same as *const_Vector* or *const_Matrix*. An implementation is permitted to prevent instantiation for other choices.

Returns: The result of the application of accumulation operator *add* with base value $T()$ applied to v divided by $v.size()$.

Note: Some specializations correspond to the VSIPL functions *vsip_vmeanval_f*, *vsip_mmeanval_f*, *vsip_cvmeanval_f*, and *vsip_cmmeanval_f*. Users should be aware of the types in the division: $T/v.size()$.

```
template <typename                                T,
         template <typename, typename> class const_View,
         typename                                Block>
T
meansqval(
    const_View<T, Block> v)
VSIP_NOTHROW;
```

```
template <typename                                T,
         template <typename, typename> class const_View,
         typename                                Block>
T
meansqval(
    const_View<complex<T>, Block> v)
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as *scalar_f* or *cscalar_f* and *const_View* the same as *const_Vector* or *const_Matrix*. An implementation is permitted to prevent instantiation for other choices.

Returns: Let set S contain, for every value *val* in v , $\text{mag}(\text{val}) * \text{mag}(\text{val})$. Returns the result of the application of accumulation operator *add* with base value $T()$ applied to the set S divided by $v.size()$.

Note: Some specializations correspond to the VSIPL functions `vsip_vmeansqval_f`, `vsip_mmeansqval_f`, `vsip_cvmeansqval_f`, and `vsip_cmmeansqval_f`. Users should be aware of the types in the division: `T / v.size()`.

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
sumval(const_View<T, Block> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have `T` the same as `scalar_f`, `scalar_i`, `cscalar_f`, or `cscalar_i` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Returns: The result of the application of accumulation operator `add` with base value `T()`.

Note: Some specializations correspond to the VSIPL functions `vsip_vsumval_f`, `vsip_msumval_f`, `vsip_vsumval_i`, `vsip_msumval_i`, `vsip_cvsumval_i`, `vsip_cmsumval_i`, `vsip_cvsumval_f`, and `vsip_cmsumval_f`.

```
template <template <typename, typename> class const_View,
         typename Block>
length_type
sumval(const_View<bool, Block> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Returns: The number of `true` values in `v`.

Note: Some specializations correspond to the VSIPL functions `vsip_vsumval_bl` and `vsip_msumval_bl`.

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
sumsqval(const_View<T, Block> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Returns: Let set S contain, for every value `val` in `v`, `val * val`. Returns the result of the application of accumulation operator `add` with base value `T()` applied to the set S .

Note: Some specializations correspond to the VSIPL functions `vsip_vsumsqval_f` and `vsip_msumsqval_f`.

8.3.9 Reduction functions also returning indices [math.fns.reductidx]

- 1 These reduction functions return both a scalar value and its corresponding index in the given view.

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
maxval (
    const_View<T, Block> v,
    Index<const_View<T, Block>::dim>& idx)
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<T, Block>::dim>` such that `v.get(idx[0], ..., idx[const_View<T, Block>::dim-1]) == res`.

Returns: The result `res` of the application of accumulation operator `max` with base value `T()`.

Note: Some specializations correspond to the VSIPL functions `vsip_vmaxval_f` and `vsip_mmaxval_f`.

```
template <typename T,
```

```

        template <typename, typename> class const_View,
        typename                               Block>
T
maxmgsqval (
    const_View<complex<T>, Block>                v,
    Index<const_View<complex<T>, Block>::dim>& idx) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported have T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<complex<T>, Block>::dim>` such that `v.get(idx[0], ..., idx[const_View<complex<T>, Block>::dim-1]) == res`.

Returns: Let S contain, for every value `val` in `v`, `mag(val) * mag(val)`. Return the result `res` of the application of accumulation operator `max` with base value `T()` applied to the set S .

Note: Some specializations correspond to the VSIPL functions `vsip_vmaxmgsqval_f` and `vsip_mmaxmgsqval_f`.

```

template <typename                               T,
        template <typename, typename> class const_View,
        typename                               Block>
T
maxmgval (
    const_View<complex<T>, Block>                v,
    Index<const_View<complex<T>, Block>::dim>& idx) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported have T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<complex<T>, Block>::dim>` such that `v.get(idx[0], ..., idx[View<complex<T>, Block>::dim-1]) == res`.

Returns: Let S contain, for every value `val` in `v`, `mag(val)`. Return the result `res` of the application of accumulation operator `max` with base value `T()` applied to the set S .

Note: Some specializations correspond to the VSIPL functions `vsip_vmaxmgval_f` and `vsip_mmaxmgval_f`.

```
template <typename                               T,
          template <typename, typename> class const_View,
          typename                               Block>
T
minval (
    const_View<T, Block>                        v,
    Index<const_View<T, Block>::dim>& idx)
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have `T` the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<T, Block>::dim>` such that `v.get(idx[0], ..., idx[View<T, Block>::dim-1]) == res`.

Returns: The result `res` of the application of accumulation operator `min` with base value `T()`.

Note: Some specializations correspond to the VSIPL functions `vsip_vminval_f` and `vsip_mminval_f`.

```
template <typename                               T,
          template <typename, typename> class const_View,
          typename                               Block>
T
minmgsqval (
    const_View<complex<T>, Block>              v,
    Index<const_View<complex<T>, Block>::dim>& idx) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have `T` the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<complex<T>, Block>::dim>` such

that `v.get(idx[0], ..., idx[const_View<complex<T>, Block>::dim-1]) == res`.

Returns: Let S contain, for every value val in v , $\text{mag}(val) * \text{mag}(val)$. Return the result res of the application of accumulation operator `min` with base value `T()` applied to the set S .

Note: Some specializations correspond to the VSIPL functions `vsip_vminmgsqval_f` and `vsip_mminmgsqval_f`.

```
template <typename T,
         template <typename, typename> class const_View,
         typename Block>
T
minmgval(
    const_View<complex<T>, Block> v,
    Index<const_View<complex<T>, Block>::dim>& idx) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have T the same as `scalar_f` and `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices.

Postconditions: `idx` is the lexicographically smallest `Index<const_View<complex<T>, Block>::dim>` such that `v.get(idx[0], ..., idx[const_View<complex<T>, Block>::dim-1]) == res`.

Returns: Let S contain, for every value val in v , $\text{mag}(val)$. Return the result res of the application of accumulation operator `min` with base value `T()` applied to the set S .

Note: Some specializations correspond to the VSIPL functions `vsip_vminmgval_f` and `vsip_mminmgval_f`.

8.3.10 Operators

[**math.fns.operators**]

- 1 Overloaded C++ operators provide synonyms for several VSIPL++ functions. See Tables 8.3 and 8.5. A listing for a function f incorporates all overloaded versions of f . The operator must obey all the restrictions of its synonym.
- 2 Overloaded C++ assignment operators provide syntax combining an arithmetic operation and an assignment to the left operand. See Table 8.4 for a summary. These are specified in [`view.vector.assign`], [`view.matrix.assign`], and [`view.tensor.assign`].

Table 8.3: Overloaded operators

function	operator
add	+
div	/
mul	*
neg	unary -
sub	binary -
eq	==
ge	>=
gt	>
le	<=
lt	<
ne	!=
land	&&
band	&
lnot	!
bnot	~
lor	
bor	

3 [Note: Some overloaded logical C++ operators differ according to their parameter types. See Table 8.5 for a summary.

]

```
template <typename T1,  
         typename T2>  
typename Promotion<T1, T2>::type  
operator^(T1 a, T2 b) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are `operator^(bool, bool)` and `operator^(scalar_i, scalar_i)`. An implementation is permitted to prevent instantiation of `operator^(T1, T2)` for other choices of T1 and T2.

Returns: `lxor(a, b)` if T1 and T2 are both `bool`. Otherwise, `bxor<T1, T2>(a, b)`.

Table 8.4: Overloaded assignment operators

operation	operator
add and assign	<code>+=</code>
div and assign	<code>/=</code>
mul and assign	<code>*=</code>
sub and assign	<code>-=</code>
& and assign	<code>&=</code>
and assign	<code> =</code>
^ and assign	<code>^=</code>

Table 8.5: Overloaded logical operators differing according to parameter types

function	operator
<code>lxor, bxor</code>	<code>^</code>

8.4 Matrix and Vector Operations

[math.matvec]

```

namespace vsip {
    // dot products
    // [math.matvec.dot]
    template <typename T0, typename T1,
              typename Block0, typename Block1>
    typename Promotion<complex<T0>, complex<T1> >::type
    cvjdot(const_Vector<complex<T0>, Block0>,
           const_Vector<complex<T1>, Block1>) VSIP_NOTHROW;

    template <typename T0, typename T1,
              typename Block0, typename Block1>
    typename Promotion<T0, T1>::type
    dot(const_Vector<T0, Block0>,
        const_Vector<T1, Block1>) VSIP_NOTHROW;

    // Transpositions
    // [math.matvec.transpose]
    template <typename T, typename Block>
    const_Matrix<T, unspecified>
    trans(const_Matrix<T, Block>) VSIP_NOTHROW;

    template <typename T, typename Block>

```

```
const_Matrix<complex<T>, unspecified>
herm(const_Matrix<complex<T>, Block>) VSIP_NOTHROW;

// Kronecker tensor product
// [math.matvec.kron]
template <typename T0, typename T1, typename T2,
         typename Block1, typename Block2>
const_Matrix<typename Promotion<T0, T2>::type, unspecified>
kron(T0,
     const_View<T1, Block1>,
     const_View<T2, Block2>) VSIP_NOTHROW;

// Outer product
// [math.matvec.outer]
template <typename T0, typename T1, typename T2,
         typename Block1, typename Block2>
const_Matrix<typename Promotion<T0, T2>::type, unspecified>
outer(T0,
     const_Vector<T1, Block1>,
     const_Vector<T2, Block2>) VSIP_NOTHROW;

// Matrix products
// [math.matvec.product]
template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspecified>
prod(const_Matrix<T0, Block0>,
     const_Matrix<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Vector<typename Promotion<T0, T1>::type, unspecified>
prod(const_Matrix<T0, Block0>,
     const_Vector<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Vector<typename Promotion<T0, T1>::type, unspecified>
prod(const_Vector<T0, Block0>,
     const_Matrix<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspecified>
prod3(const_Matrix<T0, Block0>,
     const_Matrix<T1, Block1>) VSIP_NOTHROW;
```

```

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Vector<typename Promotion<T0, T1>::type, unspecified>
prod3(const_Matrix<T0, Block0>,
      const_Vector<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspecified>
prod4(const_Matrix<T0, Block0>,
      const_Matrix<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Vector<typename Promotion<T0, T1>::type, unspecified>
prod4(const_Matrix<T0, Block0>,
      const_Vector<T1, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<complex<T0>, complex<T1> >::type,
unspecified>
prodh(const_Matrix<complex<T0>, Block0>,
      const_Matrix<complex<T1>, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<complex<T0>, complex<T1> >::type,
unspecified>
prodj(const_Matrix<complex<T0>, Block0>,
      const_Matrix<complex<T1>, Block1>) VSIP_NOTHROW;

template <typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspecified>
prodt(const_Matrix<T0, Block0>,
      const_Matrix<T1, Block1>) VSIP_NOTHROW;

// Generalized Matrix operations
// [math.matvec.gem]
template <mat_op_type OpA, mat_op_type OpB,
         typename T0, typename T1, typename T2,
         typename T3, typename T4,
         typename Block1, typename Block2,
         typename Block4>
void gemp(T0, const_Matrix<T1, Block1>,
         const_Matrix<T2, Block2>, T3,
         Matrix<T4, Block4>) VSIP_NOTHROW;

```

8.4.1. [MATH.MATVEC.DOT] DOT PRODUCTS

```
template <mat_op_type OpA, mat_op_type OpB,
         typename T0, typename T1,
         typename T3, typename T4,
         typename Block1, typename Block4>
void gems(T0, const_Matrix<T1, Block1>,
         T3, Matrix<T4, Block4>) VSIP_NOTHROW;

// Vector-Matrix products
// [math.matvec.vmmul]
template <dimension_type d,
         typename T0, typename T1,
         typename Block0, typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspecified>
vmmul(const_Vector<T0, Block0>,
      const_Matrix<T1, Block1>) VSIP_NOTHROW;

// Miscellaneous functions
// [math.matvec.misc]
template <dimension_type d,
         typename T0, typename T1,
         template <typename, typename> class const_View,
         template <typename, typename> class View,
         typename Block0, typename Block1>
void
cumsum(const_View<T0, Block0>, View<T1, Block1>)
      VSIP_NOTHROW;

template <typename T0, typename T1, typename T2,
         typename T3, typename Block0, typename Block1>
T1
modulate(const_Vector<T0, Block0>, T1, T2,
        Vector<complex<T3>, Block1>) VSIP_NOTHROW;
}
```

- 1 Behavior for domain, range, overflow, and underflow errors and for undefined values conforms to VSIPL 1.1 API, i.e., the result is undefined behavior.

8.4.1 Dot products

[**math.matvec.dot**]

```
template <typename T0,
         typename T1,
         typename Block0,
         typename Block1>
typename Promotion<complex<T0>, complex<T1> >::type
cvjdot (
```

```
const_Vector<complex<T0>, Block0> v,
const_Vector<complex<T1>, Block1> w) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported has T_0 and T_1 both equal to `scalar_f`. An implementation is permitted to prevent instantiation of `cvjdot<T0, T1, Block0, Block1>` for other choices of T_0 and T_1 . v and w must be element-conformant.

Returns: The conjugate dot product $v^T w^*$.

Note: Using T_0 or T_1 equal to `scalar_f` implies a `const_Vector<complex<scalar_f>, Block>` argument. Some specializations correspond to VSIP function `vsip_cvjdot_f`.

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
typename Promotion<T0, T1>::type
dot (
    const_Vector<T0, Block0> v,
    const_Vector<T1, Block1> w) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported have both T_0 and T_1 equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation of `dot<T0, T1, Block0, Block1>` for other choices of T_0 and T_1 . v and w must be element-conformant.

Returns: The inner dot product $v^T w$.

Note: Some specializations correspond to VSIP functions `vsip_vdot_f` and `vsip_cvdot_f`.

8.4.2 Matrix transpositions

[[math.matvec.transpose](#)]

```
template <typename T,
          typename Block>
const_Matrix<T, unspecified>
trans (const_Matrix<T, Block> m) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported has T equal to `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of T .

Returns: The transposition of m , i.e., m^T .

Note: Some specializations correspond to VSIPL functions `vsip_mtrans_f` and `vsip_cmtrans_f`. The returned matrix's block type is not necessarily equal to `Block`. VSIPL functions support in-place transposition on square matrixes, but VSIPL++ does not support in-place transposition. See [view.matrix.subviews] for an alternate transposition implementation.

```
template <typename T,
          typename Block>
const_Matrix<complex<T>, unspecified>
herm(const_Matrix<complex<T>, Block> m) VSIP_NOTHROW;
```

Requires: The only specialization which must be supported has T equal to `scalar_f`. An implementation is permitted to prevent instantiation for other choices of T .

Returns: The Hermitian (conjugate transpose) of m , i.e., m^H .

Note: Using `T` equal to `scalar_f` implies a `const_Matrix<complex<scalar_f>, Block>` argument. Some specializations correspond to VSIPL function `vsip_cmherm_f`. The returned matrix's block type is not necessarily equal to `Block`. VSIPL functions support in-place transposition on square matrixes, but VSIPL++ does not support in-place transposition.

8.4.3 Kronecker tensor product

[math.matvec.kron]

```
template <typename T0,
          typename T1,
          typename T2,
          template <typename, typename> class const_View,
          typename Block1,
          typename Block2>
const_Matrix<
    typename Promotion<T0, typename Promotion<T1, T2>::type>::type,
    unspecified>
kron(
    T0 alpha,
    const_View<T1, Block1> v,
    const_View<T2, Block2> w)
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0`, `T1`, and `T2` all the same and equal to either `scalar_f` or `cscalar_f` and `const_View` equal to either `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices of `T0`, `T1`, `T2`, and `View`.

Returns: The alpha-scaled Kronecker tensor product of `v` and `w`.

Postconditions: For `const_View` equal to `const_Vector`, the resulting matrix has `m` rows and `n` columns if `v.size() == n` && `w.size() == m`. For `const_View` equal to `const_Matrix`, the resulting matrix has `v.size(0) * w.size(0)` rows and `v.size(1) * w.size(1)` columns.

Note: Some specializations correspond to VSIPL functions `vsip_vkron_f`, `vsip_cvkron_f`, `vsip_mkron_f`, and `vsip_cmkron_f`. For a mathematical description of the product, see the VSIPL 1.1 API. The returned matrix's block's type is not necessarily equal to either `Block1` or `Block2`.

8.4.4 Outer product

[`math.matvec.outer`]

```
template <typename T0,
          typename T1,
          typename T2,
          typename Block1,
          typename Block2>
const_Matrix<
    typename Promotion<T0, typename Promotion<T1, T2>::type>::type,
    unspecified>
outer (
    T0 alpha,
    const_Vector<T1, Block1> v,
    const_Vector<T2, Block2> w)
VSIP_NO_THROW;
```

Requires: The only specializations which must be supported are those having `T0`, `T1`, and `T2` all the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0`, `T1`, and `T2`.

Returns: The alpha-scaled outer product of `v` and `w`. If both `T1` and `T2` are `cscalar_f`, the outer product is vw^H so `alpha * vwH` is returned. Otherwise, the outer product is vw^T so `alpha * vwT` is returned.

Postconditions: The resulting matrix has m rows and n columns if `v.size() == m` && `w.size() == n`.

Note: Some specializations correspond to VSIPL functions `vsip_vouter_f` and `vsip_cvouter_f`. For a mathematical description of the product, see the VSIPL 1.1 API. The returned matrix's block's type is not necessarily equal to either `Block1` or `Block2`.

8.4.5 Matrix products

[**math.matvec.product**]

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Matrix<typename Promotion<T0, T1>::type, unspeci-
fied>
prod(
    const_Matrix<T0, Block0> m0,
    const_Matrix<T1, Block1> m1) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(1) == m1.size(0)`.

Returns: The matrix product of `m0` and `m1`.

Postconditions: The resulting matrix has m rows and n columns if `m0.size(0) == m` && `m1.size(1) == n`.

Note: Some specializations correspond to VSIPL functions `vsip_mprod_f` and `vsip_cmprod_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Vector<typename Promotion<T0, T1>::type,
             unspecified>
prod(
    const_Matrix<T0, Block0> m,
    const_Vector<T1, Block1> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m.size(1) == v.size()`.

Returns: The matrix-vector product of `m` and `v`.

Postconditions: The resulting vector `res` has `res.size() == m.size(0)`.

Note: Some specializations correspond to VSIPL functions `vsip_mvprod_f` and `vsip_cmvprod_f`. The returned matrix's block's type is not necessarily equal to `Block0` or `Block1`.

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Vector<typename Promotion<T0, T1>::type,
             unspecified>
prod(
    const_Vector<T0, Block0> v,
    const_Matrix<T1, Block1> m) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m.size(0) == v.size()`.

Returns: The vector-matrix product of `v` and `m`, i.e., $v^T m$.

Postconditions: The resulting vector `res` has `res.size() == m.size(1)`.

Note: Some specializations correspond to VSIPL functions `vsip_vmprod_f` and `vsip_cvmprod_f`. The returned matrix's block's type is not necessarily equal to `Block0` or `Block1`.

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Matrix<typename Promotion<T0, T1>::type,
             unspecified>
prod3(
```

```
const_Matrix<T0, Block0> m0,  
const_Matrix<T1, Block1> m1) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(0) == m0.size(1) == m1.size(0) == 3`.

Returns: The matrix product of `m0` and `m1`.

Postconditions: The resulting matrix has three rows and `n` columns if `m1.size(1) == n`.

Note: Some specializations correspond to VSIPL functions `vsip_mprod3_f` and `vsip_cmprod3_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```
template <typename T0,  
         typename T1,  
         typename Block0,  
         typename Block1>  
const_Vector<typename Promotion<T0, T1>::type,  
            unspecified>
```

prod3(

```
const_Matrix<T0, Block0> m0,  
const_Vector<T1, Block1> v) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(0) == m0.size(1) == v.size() == 3`.

Returns: The matrix-vector product of `m0` and `v`.

Postconditions: The resulting vector has three rows.

Note: Some specializations correspond to VSIPL functions `vsip_mvprod3_f` and `vsip_cmvprod3_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```
template <typename T0,  
         typename T1,  
         typename Block0,  
         typename Block1>  
const_Matrix<typename Promotion<T0, T1>::type,
```

unspecified>

```

prod4 (
    const_Matrix<T0, Block0> m0,
    const_Matrix<T1, Block1> m1) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(0) == m0.size(1) == m1.size(0) == 4`.

Returns: The matrix product of `m0` and `m1`.

Postconditions: The resulting matrix has four rows and `n` columns if `m1.size(1) == n`.

Note: Some specializations correspond to VSIPL functions `vsip_mprod4_f` and `vsip_cmprod4_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```

template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Vector<typename Promotion<T0, T1>::type,
            unspecified>

```

```

prod4 (
    const_Matrix<T0, Block0> m0,
    const_Vector<T1, Block1> v) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(0) == m0.size(1) == v.size() == 4`.

Returns: The matrix-vector product of `m0` and `v`.

Postconditions: The resulting vector has four rows.

Note: Some specializations correspond to VSIPL functions `vsip_mvprod4_f` and `vsip_cmvprod4_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```

template <typename T0,
          typename T1,
          typename Block0,

```

```

        typename Block1>
const_Matrix<typename Promotion<complex<T0>, complex<T1> >::type,
            unspecified>

```

```

prodh(
    const_Matrix<complex<T0>, Block0> m0,
    const_Matrix<complex<T1>, Block1> m1) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having $T0$ and $T1$ both equal to `scalar_f`. An implementation is permitted to prevent instantiation for other choices of $T0$ and $T1$. `m0.size(1) == m1.size(1)`.

Returns: The matrix product of $m0$ and the Hermitian of $m1$, i.e., $m0m1^H$.

Postconditions: The resulting matrix has m rows and n columns if `m0.size(0) == m` && `m1.size(0) == n`.

Note: Using $T0$ or $T1$ equal to `scalar_f` implies a `const_Matrix<complex<scalar_f>, Block>` argument. Some specializations correspond to VSIP function `vsip_cmprodh_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```

template <typename T0,
        typename T1,
        typename Block0,
        typename Block1>
const_Matrix<typename Promotion<complex<T0>, complex<T1> >::type,
            unspecified>

```

```

prodj(
    const_Matrix<complex<T0>, Block0> m0,
    const_Matrix<complex<T1>, Block1> m1) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having $T0$ and $T1$ both equal to `scalar_f`. An implementation is permitted to prevent instantiation for other choices of $T0$ and $T1$. `m0.size(1) == m1.size(0)`.

Returns: The matrix product of $m0$ and the conjugate of $m1$, i.e., $m0m1^*$.

Postconditions: The resulting matrix has m rows and n columns if `m0.size(0) == m` && `m1.size(1) == n`.

Note: Using $T0$ or $T1$ equal to `scalar_f` implies a `const_Matrix<complex<scalar_f>, Block>` argument. Some

specializations correspond to VSIPPL function `vsip_cmprodj_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1`.

```
template <typename T0,
          typename T1,
          typename Block0,
          typename Block1>
const_Matrix<typename Promotion<T0, T1>::type,
            unspecified>
prod (
    const_Matrix<T0, Block0> m0,
    const_Matrix<T1, Block1> m1) VSIP_NO_THROW;
```

Requires: The only specializations which must be supported are those having `T0` and `T1` both the same and equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `m0.size(1) == m1.size(1)`.

Returns: The matrix product of `m0` and the transpose of `m1`, i.e., $m_0 m_1^T$.

Postconditions: The resulting matrix has `m` rows and `n` columns if `m0.size(0) == m` && `m1.size(0) == n`.

Note: Some specializations correspond to VSIPPL functions `vsip_mprod_f` and `vsip_cmprod_f`. The returned matrix's block is not necessarily equal to either `Block0` or `Block1`.

8.4.6 Generalized Matrix operations

[math.matvec.gem]

- 1 Generalized matrix operations include computing the generalized product of two matrices with accumulation and calculating the matrix sum.
- 2 If `OpX` is a `mat_op_type` value and `X` is a `Matrix`, let `OpX(X)` denote the matrix resulting from applying the `mat_op_type` operation to `X`. For example, `mat_trans(X)` denotes the matrix transpose of `X`.
- 3 `mat_ntrans` indicates no matrix operation. `mat_trans` indicates matrix transposition. `mat_herm` indicates the Hermitian conjugate. `mat_conj` indicates element-wise complex conjugation.

```
template <mat_op_type OpA,
```

```
        mat_op_type OpB,
        typename T0,
        typename T1,
        typename T2,
        typename T3,
        typename T4,
        typename Block1,
        typename Block2,
        typename Block4>
void
gemp (
    T0                alpha,
    const_Matrix<T1, Block1> A,
    const_Matrix<T2, Block2> B,
    T3                beta,
    Matrix<T4, Block4> C)
VSIP_NOTHROW;
```

Requires: OpA must equal `mat_ntrans` or `mat_trans` unless T1 is a complex type. OpB must equal `mat_ntrans` or `mat_trans` unless T2 is a complex type. OpA(A) and OpB(B) must be product-conformant. `OpA(A).size(0) == C.size(0) . OpB(B).size(1) == C.size(1) . C.block()` must not overlap either `A.block()` or `B.block()`. The only specializations which must be supported are those having T0, T1, T2, T3, and T4 all equal and equaling either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation with other template arguments.

Effects: Equivalent to `C = alpha * OpA(A) OpB(B) + beta * C`.

Note: Some specializations correspond to VSIP functions `vsip_gemp_f` and `vsip_cgemp_f`.

```
template <mat_op_type OpA,
        typename T0,
        typename T1,
        typename T3,
        typename T4,
        typename Block1,
        typename Block4>
void
gems (
```

```

T0                alpha,
const_Matrix<T1, Block1> A,
T3                beta,
Matrix<T4, Block4> C) VSIP_NOTHROW;

```

Requires: `OpA` must equal `mat_ntrans` or `mat_trans` unless `T1` is a complex type. `OpA(A)` and `C` must be element-conformant. `C.block()` must not overlap `A.block()`. The only specializations which must be supported are those having `T0`, `T1`, `T3`, and `T4` all equal and equaling either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation with other template arguments.

Effects: Equivalent to `C = alpha * OpA(A) + beta * C`.

Note: Some specializations correspond to VSIPL functions `vsip_gems_f` and `vsip_cgems_f`.

8.4.7 Vector-Matrix products

[**math.matvec.vmmul**]

```

template <dimension_type d,
         typename T0,
         typename T1,
         typename Block0,
         typename Block1>
Matrix<typename Promotion<T0, T1>::type, unspecified>
vmmul (
    const_Vector<T0, Block0> v,
    const_Matrix<T1, Block1> m)
VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having `T0` and `T1` both equal to `scalar_f`, both equal to `cscalar_f`, or `T0` equal to `scalar_f` and `T1` equal to `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T0` and `T1`. `v.size() == m.size(0)` if `d == 1`. `v.size() == m.size(1)` if `d == 0`.

Returns: The element-wise multiplication of `m` and a replication of `v`. If `d == 0`, the replication of `v` is a matrix element-conformant with `m` and having each row equal to `v`. If `d == 1`, the replication of `v` is a matrix element-conformant with `m` and having each column equal to `v`.

Note: Some specializations correspond to VSIPL functions `vsip_vmmul_f`, `vsip_cvmmul_f`, and `vsip_rvmmul_f`. The returned matrix's block is not necessarily equal to either `Block0` or `Block1`.

8.4.8 *Miscellaneous functions***[math.matvec.misc]**

```

template <dimension_type          d,
          typename                T0,
          typename                T1,
          template <typename, typename> class const_View,
          template <typename, typename> class View,
          typename                Block0,
          typename                Block1>

```

```
void
```

```
cumsum (
```

```
    const_View<T0, Block0> v,
    View<T1, Block1>      w) VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are those having `const_View` and `View` the same as `const_Vector` and `Vector` or `const_Matrix` and `Matrix`, respectively. Also, these specializations also have `T0` and `T1` equal `scalar_f`, `scalar_i`, `cscalar_f`, or `cscalar_i`. An implementation is permitted to prevent instantiation for other choices of `View`, `T0`, and `T1`. `v` and `w` must be element-conformant. `Block1` must be modifiable. Values with type `T0` must be assignable to type `T1`. `v.block()` and `w.block()` must be the same or not overlap.

Effects: `w` has values equaling the cumulative sum of values in `v`. If `View` is `Vector`, `d` is ignored and, for $0 \leq i < v.size()$, `w.get(i)` equals the sum over $0 \leq j \leq i$ of `v.get(j)`. If `View` is `Matrix` and `d == 0`, then, for $0 \leq m < v.size(0)$ and $0 \leq i < v.size(1)$, `w.get(m, i)` equals the sum over $0 \leq j \leq i$ of `v.get(m, j)`. If `View` is `Matrix` and `d == 1`, then, for $0 \leq i < v.size(0)$ and $0 \leq n < v.size(1)$, `w.get(i, n)` equals the sum over $0 \leq j \leq i$ of `v.get(j, n)`.

Note: Some specializations correspond to VSIPL functions `vsip_vcumsum_f`, `vsip_vcumsum_i`, `vsip_cvcumsum_f`, `vsip_cvcumsum_i`, `vsip_mcumsum_f`, `vsip_mcumsum_i`, `vsip_cmcumsum_f`, and `vsip_cmcumsum_i`.

```

template <typename T0,
          typename T1,
          typename T2,
          typename T3,
          typename Block0,

```

```

        typename Block1>
T1
modulate(
    const_Vector<T0, Block0>    v,
    T1                          nu,
    T2                          phi,
    Vector<complex<T3>, Block1> w) VSIP_NOTHROW;

```

Requires: The only specializations which must be supported are those having T0, T1, T2, and T3 all `scalar_f` or alternatively T0 the same as `cscalar_f` and T1, T2, and T3 all `scalar_f`. An implementation is permitted to prevent instantiation for other choices of T0, T1, T2, and T3. `v` and `w` must be element-conformant. `Block1` must be modifiable. T0, T1, T2, and T3 must be such that $t_0 * \cos(k * t_1 + t_2)$ can be assigned to a T3 value, where values `t0`, `k`, `t1`, and `t2` have types T0, `index_type`, T1, and T2, respectively. `v.block()` and `w.block()` must either be the same or not overlap.

Effects: For $0 \leq i < v.size()$, `w.get(i)` has a value equaling the product of `v.get(i)` and the exponential of the product of $i(\sqrt{-1})$ and $i * nu + phi$.

Returns: `v.size() * nu + phi`.

Note: Some specializations correspond to VSIPL functions `vsip_vmodulate_f` and `vsip_cvmodulate_f`.

8.5 Linear system solvers

[**math.solvers**]

- 1 These functions and classes solve linear systems and also perform singular value decomposition.
- 2 [*Note:* Many of these solvers throw exceptions, e.g., `std::bad_alloc` and `computation_error`, to indicate memory allocation and computation errors.]

Header <vsip/solvers.hpp> synopsis

```

namespace vsip {
    // solve covariance linear system
    template <typename T, typename Block0, typename Block1>
    const_Matrix<T, unspecified>
    covsol(Matrix<T, Block0>, const_Matrix<T, Block1>)
    VSIP_THROW((std::bad_alloc, computation_error));
}

```

```
template <typename T, typename Block0, typename Block1,
         typename Block2>
Matrix<T, Block2>&
covsol(Matrix<T, Block0>, const_Matrix<T, Block1>,
        Matrix<T, Block2>)
VSIP_THROW((std::bad_alloc, computation_error));

// linear least squares solver
template <typename T, typename Block0, typename Block1>
const_Matrix<T, unspecified>
llsqsol(Matrix<T, Block0>, const_Matrix<T, Block1>)
VSIP_THROW((std::bad_alloc, computation_error));

template <typename T, typename Block0, typename Block1,
         typename Block2>
Matrix<T, Block2>
llsqsol(Matrix<T, Block0>, const_Matrix<T, Block1>,
        Matrix<T, Block2>)
VSIP_THROW((std::bad_alloc, computation_error));

// Toeplitz linear system solver
template <typename T, typename Block0, typename Block1,
         typename Block2>
const_Vector<T, unspecified>
toepsol(const_Vector<T, Block0>, const_Vector<T, Block1>,
        Vector<T, Block2>)
VSIP_THROW((std::bad_alloc, computation_error));

template <typename T, typename Block0, typename Block1,
         typename Block2, typename Block3>
Vector<T, Block3>
toepsol(const_Vector<T, Block0>, const_Vector<T, Block1>,
        Vector<T, Block2>, Vector<T, Block3>)
VSIP_THROW((std::bad_alloc, computation_error));

// LU decomposition linear system solver
template <typename T = VSIP_DEFAULT_VALUE_TYPE,
         return_mechanism_type ReturnMechanism = by_value>
class lud;

// Cholesky decomposition linear system solver
template <typename T = VSIP_DEFAULT_VALUE_TYPE,
         return_mechanism_type ReturnMechanism = by_value>
class chold;
```

```

// QR decomposition linear system solver
template <typename T = VSIP_DEFAULT_VALUE_TYPE,
         return_mechanism_type ReturnMechanism = by_value>
class qrd;

// singular value decomposition
template <typename T = VSIP_DEFAULT_VALUE_TYPE,
         return_mechanism_type ReturnMechanism = by_value>
class svd;
}

```

8.5.1 Covariance linear system solver

[math.solvers.covsol]

```

template <typename T,
         typename Block0,
         typename Block1>
const_Matrix<T, unspecified>
covsol (
    Matrix<T, Block0> m0,
    const_Matrix<T, Block1> m1)
VSIP_THROW((std::bad_alloc, computation_error));

```

Requires: $m0.size(0) \geq m0.size(1)$. $m0.size(1) == m1.size(0)$. The rank of $m0$ must equal $m0.size(1)$. $m0$ and $m1$ must not overlap. The only required specializations which must be supported are for T equal to either *scalar_f* or *cscalar_f* . An implementation is permitted to prevent instantiations for other choices of T . $Block0$ must be modifiable.

Returns: The solution X to the covariance linear system $m0^T m0 X = m1$ if T is *scalar_f* . The solution X to the covariance linear system $m0^H m0 X = m1$ if T is *cscalar_f* .

Postconditions: The returned matrix is element-conformant with $m1$. $m0$ may have been overwritten.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if $m0$ does not have full column rank.

Note: Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines. This function corresponds to VSIP functions *vsip_covsol_f* and *vsip_ccovsol_f* . The returned matrix's block's type is not necessarily equal to either $Block0$ or $Block1$.

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2>
Matrix<T, Block2>
covsol (
    Matrix<T, Block0>      m0,
    const_Matrix<T, Block1> m1,
    Matrix<T, Block2>     m2)
VSIP_THROW((std::bad_alloc, computation_error));
```

Requires: $m0.size(0) \geq m0.size(1)$. $m0.size(1) == m1.size(0)$. The rank of $m0$ must equal $m0.size(1)$. $m1$ and $m2$ are element-conformant. $m0$, $m1$, and $m2$ must not overlap. $Block0$ and $Block2$ must be modifiable. The only required specializations which must be supported are for T equal to either `scalar_f` or `cscalar_f` . An implementation is permitted to prevent instantiations for other choices of T .

Effects: The solution X to the covariance linear system $m0^T m0 X = m1$ is stored in $m2$ if T is `scalar_f` . The solution X to the covariance linear system $m0^H m0 X = m1$ is stored in $m2$ if T is `cscalar_f` .

Returns: $m2$.

Postconditions: $m0$ may have been overwritten.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if $m0$ does not have full column rank.

Note: Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines. This function corresponds to VSIPL functions `vsip_covsol_f` and `vsip_ccovsol_f`.

8.5.2 Linear least squares solver

[[math.solvers.llsqsol](#)]

```
template <typename T,
          typename Block0,
          typename Block1>
Matrix<T, unspecified>
llsqsol (
    Matrix<T, Block0>      m0,
    const_Matrix<T, Block1> m1)
VSIP_THROW((std::bad_alloc, computation_error));
```

Requires: `m0.size(0) >= m0.size(1)` . `m0.size(0) == m1.size(0)` . The rank of `m0` must equal `m0.size(1)` . `m0` and `m1` must not overlap. The only required specializations which must be supported are for `T` equal to either `scalar_f` or `cscalar_f` . An implementation is permitted to prevent instantiations for other choices of `T` . `Block0` must be modifiable.

Returns: The solution X to the linear least squares problem $\min_X \|m_0 X - m_1\|_2$.

Postconditions: The returned matrix has `m0.size(1)` rows and `m1.size(1)` columns. `m0` may have been overwritten.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if `m0` does not have full column rank.

Note: Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines. This function corresponds to VSIPL functions `vsip_llsqsol_f` and `vsip_cllsqsol_f`. The returned matrix's block's type is not necessarily equal to either `Block0` or `Block1` .

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2>
Matrix<T, Block2>
llsqsol (
    Matrix<T, Block0>      m0,
    const_Matrix<T, Block1> m1,
    Matrix<T, Block2>      m2)
VSIP_THROW((std::bad_alloc, computation_error));
```

Requires: `m0.size(0) >= m0.size(1)` . `m0.size(0) == m1.size(0)` . The rank of `m0` must equal `m0.size(1)` . `m2.size(0) == m0.size(1)` . `m2.size(1) == m1.size(1)` . `m0`, `m1`, and `m2` must not overlap. `Block0` and `Block2` must be modifiable. The only required specializations which must be supported are for `T` equal to either `scalar_f` or `cscalar_f` . An implementation is permitted to prevent instantiations for other choices of `T` .

Effects: The solution X to the linear least squares problem $\min_X \|m_0 X - m_1\|_2$ is placed in `m2` .

Returns: `m2` .

Postconditions: `m0` may have been overwritten.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if `m0` does not have full column rank.

Note: Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines. This function corresponds to VSIPL functions `vsip_llsqsol_f` and `vsip_cllsqsol_f`.

8.5.3 *Toeplitz linear system solver***[math.solvers toeplitz]**

```
template <typename T,
         typename Block0,
         typename Block1,
         typename Block2>
const_Vector<T, unspecified>
toepsol (
    const_Vector<T, Block0> t,
    const_Vector<T, Block1> b,
    Vector<T, Block2>      w)
VSIP_THROW((std::bad_alloc, computation_error));
```

Requires: The Toeplitz matrix `T` formed from `t` must have full rank and be positive definite. `t.size() == b.size() == w.size()`. `w`'s block may not overlap `t`'s block nor `b`'s block. The only required specializations which must be supported are for `T` equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiations for other choices of `T`. `Block2` must be a modifiable type.

Returns: The solution `x` to the Toeplitz linear system $TX = b$, where `t` specifies the Toeplitz matrix `T`. The Toeplitz linear system is real symmetric if `T` is `scalar_f` and is Hermitian if `T` is `cscalar_f`.

Postconditions: The returned vector has `t.size()` entries.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if the Toeplitz matrix `T` does not have full column rank or is not positive definite.

Note: `w` might be used as a temporary workspace. This function corresponds to VSIPL functions `vsip_toepsol_f` and `vsip_ctoepsol_f`. For a mathematical description of the system, see the VSIPL 1.1 API. The returned vector's block's type is not necessarily equal to `Block0`, `Block1`, or `Block2`.

```
template <typename T,
         typename Block0,
```

```

        typename Block1,
        typename Block2,
        typename Block3>
Vector<T, Block3>
toepsol(
    const_Vector<T, Block0> t,
    const_Vector<T, Block1> b,
    Vector<T, Block2>      w,
    Vector<T, Block3>      answer)
VSIP_THROW((std::bad_alloc, computation_error));

```

Requires: The Toeplitz matrix T formed from t must have full rank and be positive definite. $t.size() == b.size() == w.size() == answer.size()$. The $answer$'s block must not overlap t 's block nor b 's block nor w 's block. w 's block may not overlap t 's block nor b 's block. w and $answer$ must be modifiable. The only required specializations which must be supported are for T the equal to either `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiations for other choices of T .

Effects: The solution X to the Toeplitz linear system $TX = b$, where t specifies the Toeplitz matrix T , is placed in $answer$. The Toeplitz linear system is real symmetric if T is `scalar_f` and is Hermitian if T is `cscalar_f`.

Returns: $answer$.

Postconditions: The returned vector has $t.size()$ entries.

Throws: `std::bad_alloc` upon a memory allocation error and `computation_error` if the Toeplitz matrix T does not have full column rank or is not positive definite.

Note: w might be used as a temporary workspace. This function corresponds to VSIPL functions `vsip_toepsol_f` and `vsip_ctoepsol_f`.

8.5.4 LU linear system solver

[[math.solvers.lu](#)]

- 1 The template class `lud` uses LU (lower and upper triangular) decomposition to solve a linear system. The only specializations of `lud` which must be supported are `lud<scalar_f, RM>` and `lud<cscalar_f, RM>` for any `return_mechanism_type` value of `RM`. An implementation is permitted to prevent instantiation of `lud<T, RM>` for other choices of T .

```

namespace vsip {
    template <typename T = VSIP_DEFAULT_VALUE_TYPE,

```

```
        return_mechanism_type ReturnMechanism = by_value>
class lud {
public:
    // constructors, copies, assignment, and destructors
    lud(length_type) VSIP_THROW((std::bad_alloc));
    lud(lud const&) VSIP_THROW((std::bad_alloc));
    lud& operator=(lud const&) VSIP_NOTHROW;
    ~lud() VSIP_NOTHROW;

    // accessor
    length_type length() const VSIP_NOTHROW;

    // solve a system
    template <typename Block>
    bool decompose(Matrix<T, Block>) VSIP_NOTHROW;

    // if ReturnMechanism is by_value:
    template <mat_op_type tr, typename Block>
    Matrix<T, unspecified>
    solve(const_Matrix<T, Block>)
        VSIP_THROW((computation_error));

    // if ReturnMechanism is by_reference:
    template <mat_op_type tr, typename Block1, typename Block2>
    bool solve(const_Matrix<T, Block1>, Matrix<T, Block2>)
        VSIP_NOTHROW;
};
}
```

lud(length_type len) VSIP_THROW((std::bad_alloc));

Requires: len > 0.

Effects: Constructs an lud object that will decompose len by len matrices.

Throws: std::bad_alloc indicating memory allocation for the returned lud object failed.

Note: This function corresponds to VSIPL functions vsip_lud_create_f and vsip_clud_create_f.

length_type

length() const VSIP_NOTHROW;

Returns: The number of rows in a decomposed matrix. The number of columns is the same.

Note: The returned value equals the required number of rows in the matrix given to decompose . This function corresponds to VSIPL functions vsip_lud_getattr_f and vsip_clud_getattr_f.

```
template <typename Block>
bool
decompose(Matrix<T, Block> m) VSIP_NOTHROW;
```

Requires: *m* must be a square matrix with the same number of rows and columns and having full rank. The number of rows must equal `this->length()`. *m* must be modifiable.

Effects: Performs LU decomposition of *m*. The matrix *m* may be overwritten.

Returns: `false` if the decomposition fails.

Postconditions: The matrix *m* may not be modified as long as its decomposition may be used.

Note: The choice of using row interchanges or column interchanges is implementation dependent. This function corresponds to VSIP functions `vsip_lud_f` and `vsip_clud_f`.

```
template <mat_op_type tr,
          typename Block>
const_Matrix<T, unspecified>
solve(const_Matrix<T, Block> b)
VSIP_THROW((computation_error));
```

Requires: `b.size(0) == this->length()`. A call to `decompose` for this object must have occurred. `ReturnMechanism == by_value`.

Returns: A matrix *m* containing the solution. If `T == scalar_f` and `tr == mat_trans`, *m* is the solution to $A^T m = b$, where *A* is the matrix given to the most recent `decompose` call for this object. If `T == cscalar_f` and `tr == mat_herm`, *m* is the solution to $A^H m = b$. Otherwise, $A m = b$ is solved.

Postconditions: *m* and *b* are element-conformant.

Throws: `computation_error` if the computation fails.

Note: This function corresponds to VSIP functions `vsip_lusol_f` and `vsip_clusol_f`. The returned matrix's block's type is not necessarily equal to `Block`.

```
template <mat_op_type tr,
          typename Block1,
          typename Block2>
bool
solve(const_Matrix<T, Block1> b, Matrix<T, Block2> answer)
VSIP_NOTHROW;
```

Requires: `b.size(0) == length()` . `b` and `answer` are element-conformant and must not overlap. A call to `decompose` for this object must have occurred. `ReturnMechanism == by_reference`. `answer` must be modifiable.

Effects: A matrix `m` containing the solution is placed in `answer` . If `T == scalar_f` and `tr == mat_trans`, `m` is the solution to $A^T m = b$, where `A` is the matrix given to the most recent `decompose` call for this object. If `T == cscalar_f` and `tr == mat_herm`, `m` is the solution to $A^H m = b$. Otherwise, $A m = b$ is solved.

Returns: `true` if the computation succeeds.

Note: This function corresponds to VSIPL functions `vsip_lusol_f` and `vsip_clusol_f`.

8.5.5 Cholesky decomposition linear system solver [math.solvers.cholesky]

- The template class `chold` uses Cholesky decomposition to solve a linear system. The only specializations of `chold` which must be supported are `chold<scalar_f, RM>` and `chold<cscalar_f, RM>` for any `return_mechanism_type` value of `RM` . An implementation is permitted to prevent instantiation of `chold<T, RM>` for other choices of `T` .

```
namespace vsip {
    // enumerations
    enum mat_uplo { lower, upper };

    template <typename T = VSIP_DEFAULT_VALUE_TYPE,
              return_mechanism_type ReturnMechanism = by_value>
    class chold {
    public:
        // constructors, copies, assignments, and destructors
        chold(mat_uplo, length_type) VSIP_THROW((std::bad_alloc));
        chold(chold const&) VSIP_THROW((std::bad_alloc));
        chold& operator=(chold const&) VSIP_NOTHROW;
        ~chold() VSIP_NOTHROW;

        // accessors
        length_type length() const VSIP_NOTHROW;
        mat_uplo uplo() const VSIP_NOTHROW;

        // solve a system
        template <typename Block>
        bool decompose(Matrix<T, Block>) VSIP_NOTHROW;
    };
}
```

```

// if ReturnMechanism is by_value:
template <typename Block>
Matrix<T, unspecified>
solve(const_Matrix<T, Block>)
    VSIP_THROW((computation_error));

// if ReturnMechanism is by_reference:
template <typename Block, typename Block1>
bool solve(const_Matrix<T, Block>, Matrix<T, Block1>)
    VSIP_NOTHROW;
};
}

```

- 2 [Note: enum `mat_uplo` indicates which half of a symmetric or Hermitian matrix is referenced. `lower` indicates the lower half of the matrix is referenced. `upper` indicates the upper half of the matrix is referenced.]

```

chold(mat_uplo uplo, length_type len)
    VSIP_THROW((std::bad_alloc));

```

Requires: Positive `len`.

Effects: Constructs a `chold` object that will decompose `len` by `len` symmetric positive definite matrices.

Throws: `std::bad_alloc` indicating memory allocation for the returned `chold` object failed.

Note: This function corresponds to VSIPL functions `vsip_chold_create_f` and `vsip_cchold_create_f`.

```

length_type length() const VSIP_NOTHROW;

```

Returns: The number of rows in a decomposed matrix. The number of columns is the same.

Note: The returned value equals the required number of rows in the matrix given to decompose for this object. This function corresponds to VSIPL functions `vsip_chold_getattr_f` and `vsip_cchold_getattr_f`.

```

mat_uplo uplo() const VSIP_NOTHROW;

```

Returns: An indication whether the lower half or upper half of a decomposed matrix is referenced.

Note: This function corresponds to VSIPL functions `vsip_chold_getattr_f` and `vsip_cchold_getattr_f`.

```
template <typename Block>
bool
decompose(Matrix<T, Block> m) VSIP_NOTHROW;
```

Requires: m must be a square matrix with the same number of rows and columns. If T is *scalar_f*, m must be symmetric positive definite. If T is *cscalar_f*, m must be Hermitian positive definite. The number of rows must equal `length() . m` must be modifiable.

Effects: Performs Cholesky decomposition of m . The matrix m may be overwritten.

Returns: `false` if the decomposition fails. It will fail if a leading minor of m is not symmetric or Hermitian positive definite and the algorithm cannot complete.

Postconditions: The matrix m may not be modified as long as its decomposition may be used.

Note: This function corresponds to VSIP functions *vsip_chold_f* and *vsip_cchold_f*.

```
template <typename Block>
const_Matrix<T, unspecified>
solve(const_Matrix<T, Block> b)
VSIP_THROW((computation_error));
```

Requires: `b.size(0) == length()`. A call to `decompose` for this object must have occurred. `ReturnMechanism == by_value`.

Returns: A constant matrix m containing the solution to $Am = b$, where A is the matrix given to the most recent `decompose` call for this object.

Postconditions: m and b are element-conformant.

Throws: `computation_error` if the most recent `decompose` call for this object failed.

Note: This function corresponds to VSIP functions *vsip_cholsol_f* and *vsip_ccholsol_f*. The returned matrix's block's type is not necessarily equal to `Block`.

```
template <typename Block, typename Block1>
```

```
bool
solve(
    const_Matrix<T, Block> b,
```

```
Matrix<T, Block1>    answer)
VSIP_NOTHROW;
```

Requires: `b.size(0) == length()` . A call to `decompose` must have occurred. `b` and `answer` are element-conformant and must not overlap. `answer` must be modifiable. `ReturnMechanism == by_reference` .

Effects: A matrix `m` containing the solution to $A_m = b$, where `A` is the matrix given to the most recent `decompose` call for this object, is placed in `answer` .

Returns: `true` if the computation succeeds.

Note: This function corresponds to VSIPL functions `vsip_cholsol_f` and `vsip_ccholsol_f`.

8.5.6 QR decomposition linear system solver

[[math.solvers.qr](#)]

- 1 The template class `qrd` uses QR decomposition to decompose a matrix and solve linear systems. The only specializations of `qrd` which must be supported are `qrd<scalar_f, RM>` and `qrd<cscalar_f, RM>` for any `return_mechanism_type` value `RM` . An implementation is permitted to prevent instantiation of `qrd<T, RM>` for other choices of `T` .

```
namespace vsip {
  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           return_mechanism_type ReturnMechanism = by_value>
  class qrd {
  public:
    // constructors, copies, assignments, and destructors
    qrd(length_type, length_type, storage_type)
      VSIP_THROW((std::bad_alloc));
    qrd(qrd const&) VSIP_THROW((std::bad_alloc));
    qrd& operator=(qrd const&) VSIP_NOTHROW;
    ~qrd() VSIP_NOTHROW;

    // accessors
    length_type rows() const VSIP_NOTHROW;
    length_type columns() const VSIP_NOTHROW;
    storage_type qstorage() const VSIP_NOTHROW;

    // solve systems
    template <typename Block>
    bool decompose(Matrix<T, Block>) VSIP_NOTHROW;

    // if ReturnMechanism is by_value:
```

```

template <mat_op_type tr, product_side_type ps, typename Block>
Matrix<T, unspecified>
prodq(const_Matrix<T, Block>)
    VSIP_THROW((computation_error));

template <mat_op_type tr, typename Block>
Matrix<T, unspecified>
rsol(const_Matrix<T, Block>, T const)
    VSIP_THROW((computation_error));

template <typename Block>
Matrix<T, unspecified>
covsol(const_Matrix<T, Block>)
    VSIP_THROW((computation_error));

template <typename Block>
Matrix<T, unspecified>
lsqsol(const_Matrix<T, Block>)
    VSIP_THROW((computation_error));

// if ReturnMechanism is by_reference:
template <mat_op_type tr, product_side_type ps,
        typename Block0, typename Block1>
bool
prodq(const_Matrix<T, Block0>, Matrix<T, Block1>)
    VSIP_NOTHROW;

template <mat_op_type tr, typename Block0, typename Block1>
bool
rsol(const_Matrix<T, Block0>, T const,
     Matrix<T, Block1>) VSIP_NOTHROW;

template <typename Block0, typename Block1>
bool
covsol(const_Matrix<T, Block0>, Matrix<T, Block1>)
    VSIP_NOTHROW;

template <typename Block0, typename Block1>
bool
lsqsol(const_Matrix<T, Block0>, Matrix<T, Block1>)
    VSIP_NOTHROW;
};
}

```

- 2 [Note: Declared in [math.enum], enum `storage_type` indicates the storage format for decomposed matrixes. `qrd_nosaveq` indicates the object does not store Q . `qrd_saveq1` indicates Q is stored using the same amount of space as the matrix m given to the constructor. `qrd_saveq` indicates the square matrix Q

is stored using the same number of rows as `m`.]

```
qrd(
    length_type rows,
    length_type columns,
    storage_type st)
VSIP_THROW((std::bad_alloc));
```

Requires: `rows >= columns > 0`.

Effects: Constructs a `qrd` object.

Throws: `std::bad_alloc` indicating memory allocation for the returned `qrd` object failed.

Note: This function corresponds to VSIPL functions `vsip_qrd_create_f` and `vsip_cqrd_create_f`.

```
length_type rows() const VSIP_NOTHROW;
```

Returns: The number of rows in the Q matrix.

Note: This function corresponds to VSIPL functions `vsip_qrd_getattr_f` and `vsip_cqrd_getattr_f`.

```
length_type columns() const VSIP_NOTHROW;
```

Returns: The number of columns in the Q matrix.

Note: This function corresponds to VSIPL functions `vsip_qrd_getattr_f` and `vsip_cqrd_getattr_f`.

```
storage_type qstorage() const VSIP_NOTHROW;
```

Returns: The storage type for the Q matrix, as specified in the constructor.

Note: This function corresponds to VSIPL functions `vsip_qrd_getattr_f` and `vsip_cqrd_getattr_f`.

```
template <typename Block>
bool
decompose(Matrix<T, Block> m) VSIP_NOTHROW;
```

Requires: `m.size(0) == this->rows()` . `m.size(1) == this->columns()` . `m` must have full column rank equaling `columns()` and be modifiable.

Effects: Performs a QR decomposition of m into matrices Q and R . The matrix m may be overwritten.

Returns: `false` if the decomposition fails because m does not have full column rank.

Postconditions: The matrix m may not be modified as long as its decomposition may be used.

Note: In the decomposition, Q has `this->rows()` rows and `this->columns()` columns. If T is a specialization of `complex`, Q is unitary. Otherwise, Q is orthogonal. R is an upper triangular matrix. If m has full rank, then R is a nonsingular matrix. No column interchanges are performed. This function corresponds to VSIPL functions `vsip_qrd_f` and `vsip_cqrd_f`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename         Block>
const_Matrix<T, unspecified>
prodq(const_Matrix<T, Block> m) VSIP_THROW((computation_error));
```

Requires: A call to `decompose` must have occurred for this object with `this->qstorage()` equaling either `qrd_saveq1` or `qrd_saveq`. Otherwise, the behavior is undefined. `ReturnMechanism == by_value`. The number of rows and columns of m depends on the values of `tr`, `ps`, and `this->qstorage()`. For `this->qstorage() == qrd_saveq1`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	<code>columns(), s</code>	<code>s, rows()</code>
<code>tr == mat_trans</code>	<code>rows(), s</code>	<code>s, columns()</code>
<code>tr == mat_herm</code>	<code>rows(), s</code>	<code>s, columns()</code>

where s is an arbitrary positive `length_type`. For `qstorage() == qrd_saveq`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	<code>rows(), s</code>	<code>s, rows()</code>
<code>tr == mat_herm</code>	<code>rows(), s</code>	<code>s, rows()</code>

Returns: The product of Q and m . The actual product and its number of rows and columns depends on the values of `tr`, `ps`, and `qstorage()` and

whether `T` is not or is a specialization of `complex`. For `qstorage()` == `qrd_saveq1`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	$Qm, rows(), s$	$mQ, s, columns()$
<code>tr == mat_trans, T</code>	$Q^T m, columns(), s$	$mQ^T, s, rows()$
<code>tr == mat_herm, complex<T></code>	$Q^H m, columns(), s$	$mQ^H, s, rows()$

where `s` is the same variable as above. For `qstorage()` == `qrd_saveq`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	$Qm, rows(), s$	$mQ, s, rows()$
<code>tr == mat_trans, T</code>	$Q^T m, rows(), s$	$mQ^T, s, rows()$
<code>tr == mat_herm, complex<T></code>	$Q^H m, rows(), s$	$mQ^H, s, rows()$

Throws: `computation_error` if the product fails.

Note: This function corresponds to VSIPL functions `vsip_qrdprodq_f` and `vsip_cqrdprodq_f`. The returned matrix's block's type is not necessarily equal to `Block`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename         Block0,
         typename         Block1>
bool
prodq(
    const_Matrix<T, Block0> m,
    Matrix<T, Block1>       destination)
VSIP_NOTHROW;
```

Requires: A call to `decompose` must have occurred for this object with `qstorage()` equaling either `qrd_saveq1` or `qrd_saveq`. Otherwise, the behavior is undefined. `ReturnMechanism == by_reference`. `destination` must be modifiable. `m` and `destination` must not overlap. The number of rows and columns of `m` depends on the values of `tr`, `ps`, and `qstorage()`. For `qstorage() == qrd_saveq1`,

8.5.6. [MATH.SOLVERS.QR] QR DECOMPOSITION LINEAR SYSTEM...

	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 length_type . 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 destination depends 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()

Effects: The product of Q and m is stored in `destination` . 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	Qm	mQ
tr == mat_trans, T	$Q^T m$	mQ^T
tr == mat_herm, complex<T>	$Q^H m$	mQ^H

Returns: true if the product succeeds.

Note: This function corresponds to VSIPPL functions `vsip_qrdprodq_f` and `vsip_cqrdprodq_f`.

```

template <mat_op_type tr,
         typename Block>
const_Matrix<T, unspecified>
rsol(
    const_Matrix<T, Block> b,
    T const                alpha)
VSIP_THROW((computation_error));

```

Requires: `b.size(0) == this->columns()` . A call to decompose for this object must have occurred. ReturnMechanism == `by_value`.

Returns: A constant matrix `m` containing the solution. If `tr == mat_trans` and `T` is not a specialization of `complex`, then $R^T m = \alpha * b$ is solved. If `tr == mat_herm` and `T` is a specialization of `complex`, then $R^H m = \alpha * b$ is solved. Otherwise, $Rm = \alpha * b$ is solved.

Postconditions: `m` and `b` are element-conformant.

Throws: `computation_error` if the algorithm could not be computed.

Note: This function corresponds to VSIPL functions `vsip_qrdsolr_f` and `vsip_cqrdsolr_f`. The returned matrix's block's type is not necessarily equal to `Block`.

```

template <mat_op_type tr,
         typename Block0,
         typename Block1>
bool
rsol(
    const_Matrix<T, Block0> b,
    T const                alpha,
    Matrix<T, Block1>      destination)
VSIP_NOTHROW;

```

Requires: `b.size(0) == this->columns()` . A call to decompose for this object must have occurred. ReturnMechanism == `by_reference`. `destination` must be modifiable and element conformant with `b`. `b` and `destination` must not overlap.

Effects: Stores the solution in `destination` . If `tr == mat_trans` and `T` is not a specialization of `complex`, then $R^T \text{destination} = \alpha * b$ is solved. If `tr == mat_herm` and `T` is a specialization of `complex`, then $R^H \text{destination} = \alpha * b$ is solved. Otherwise, $R \text{destination} = \alpha * b$ is solved.

Returns: true if the algorithm can be computed.

Note: This function corresponds to VSIPL functions `vsip_qrdsolr_f` and `vsip_cqrdsolr_f`.

```
template <typename Block>
const_Matrix<T, unspecified>
covsol(const_Matrix<T, Block> b)
VSIP_THROW((computation_error));
```

Requires: `b.size(0) == this->columns()` . ReturnMechanism == `by_value` .

Returns: A matrix `m` containing the solution. If `T` is not a specialization of `complex`, then $A^T A m = b$ is solved, where A is the matrix given to the most recent call to `decompose` to this object. If `T` is a specialization of `complex`, then $A^H A m = b$ is solved.

Postconditions: `m` and `b` are element-conformant.

Throws: `computation_error` if the algorithm fails.

Note: This function corresponds to VSIPL functions `vsip_qrsol_f` and `vsip_cqrsol_f`. The returned matrix's block's type is not necessarily equal to `Block` .

```
template <typename Block0,
          typename Block1>
bool
covsol(
    const_Matrix<T, Block0> b,
    Matrix<T, Block1> destination)
VSIP_NOTHROW;
```

Requires: `b.size(0) == this->columns()` . ReturnMechanism == `by_reference` . `destination` is modifiable and is element conformant with `b` . `b` and `destination` must not overlap.

Effects: The solution is stored in `destination` . If `T` is not a specialization of `complex`, then $A^T A destination = b$ is solved, where A is the matrix given to the most recent call to `decompose` for this object. If `T` is a specialization of `complex`, then $A^H A destination = b$ is solved.

Returns: true if the algorithm succeeds.

Note: This function corresponds to VSIPL functions `vsip_qrsol_f` and `vsip_cqrsol_f`.

```
template <typename Block>
const_Matrix<T, unspecified>
lsqsol(const_Matrix<T, Block> b)
VSIP_THROW((computation_error));
```

Requires: `b.size(0) == rows()` . `ReturnMechanism == by_value` .

Returns: A constant matrix `m` containing the solution to the linear least squares problem $\min_m \|Am - b\|_2$, where A is the matrix given to the most recent call to `decompose` for this object.

Postconditions: `m.size(0) == this->columns()` .

Throws: `computation_error` if the algorithm fails.

Note: This function corresponds to VSIPL functions `vsip_qrsol_f` and `vsip_cqrsol_f`. The returned matrix's block's type is not necessarily equal to `Block` .

```
template <typename Block0,
          typename Block1>
bool
lsqsol(
    const_Matrix<T, Block0> b,
    Matrix<T, Block1> destination)
VSIP_NOTHROW;
```

Requires: `b.size(0) == this->rows()` . `destination.size(0) == this->columns()` . `ReturnMechanism == by_reference` . `b` and `destination` must not overlap.

Effects: Stores the solution to the linear least squares problem $\min_{\text{destination}} \|A_{\text{destination}} - b\|_2$, where A is the matrix given to the most recent call to `decompose` for this object, in `destination` .

Returns: `true` if the algorithm succeeds.

Note: This function corresponds to VSIPL functions `vsip_qrsol_f` and `vsip_cqrsol_f`.

8.5.7 Singular-value decomposition

[[math.solvers.svd](#)]

- 1 The template class `svd` uses singular-value decomposition to decompose a matrix into orthogonal or unitary matrixes and singular values. The only specializations of `svd` which must be supported are `svd<scalar_f, RM>` and

`svd<cscalar_f, RM>` for any `return_mechanism_type` value of `RM`. An implementation is permitted to prevent instantiation of `svd<T, RM>` for other choices of `T`.

```
namespace vsip {
    template <typename T = VSIP_DEFAULT_VALUE_TYPE,
              return_mechanism_type ReturnMechanism = by_value>
    class svd {
    public:
        // constructors, copies, assignments, and destructors
        svd(length_type, length_type, storage_type, storage_type)
            VSIP_THROW((std::bad_alloc));
        svd(svd const&) VSIP_NOTHROW;
        svd& operator=(svd const&) VSIP_NOTHROW;
        ~svd() VSIP_NOTHROW;

        // accessors
        length_type rows() const VSIP_NOTHROW;
        length_type columns() const VSIP_NOTHROW;
        storage_type ustorage() const VSIP_NOTHROW;
        storage_type vstorage() const VSIP_NOTHROW;

        // decomposition

        // if ReturnMechanism is by_value:
        template <typename Block>
        const_Vector<scalar_f, unspecified>
        decompose(Matrix<T, Block>)
            VSIP_THROW((std::bad_alloc, computation_error));

        template <mat_op_type tr, product_side_type ps, typename Block>
        const_Matrix<T, unspecified>
        produ(const_Matrix<T, Block>) const
            VSIP_THROW((computation_error));

        template <mat_op_type tr, product_side_type ps, typename Block>
        const_Matrix<T, unspecified>
        prodv(const_Matrix<T, Block>) const
            VSIP_THROW((computation_error));

        const_Matrix<T, unspecified>
        u(index_type, index_type) const VSIP_THROW((computation_error));

        const_Matrix<T, unspecified>
        v(index_type, index_type) const VSIP_THROW((computation_error));

        // if ReturnMechanism is by_reference:
        template <typename Block0, typename Block1>
```

```

bool
decompose(Matrix<T, Block0>,
          Vector<scalar_f, Block1>)
    VSIP_NOTHROW;

template <mat_op_type tr, product_side_type ps,
          typename Block0, typename Block1>
bool
produ(const_Matrix<T, Block0>, Matrix<T, Block1>)
    const VSIP_NOTHROW;

template <mat_op_type tr, product_side_type ps,
          typename Block0, typename Block1>
bool
prodv(const_Matrix<T, Block0>, Matrix<T, Block1>)
    const VSIP_NOTHROW;

template <typename Block>
bool
u(index_type, index_type, Matrix<T, Block>) const VSIP_NOTHROW;

template <typename Block>
bool
v(index_type, index_type, Matrix<T, Block>) const VSIP_NOTHROW;
};
}

```

2 Given an $m \times n$ matrix A to decompose, let $p = \min(m, n)$.

[*Note:* Declared in [math.enum], enum `storage_type` indicates the storage format for decomposed matrixes. `svd_uvnos` indicates the matrix is not stored. `svd_uvpart` indicates the first p columns of U or the first p rows of V^T or V^H are stored. `svd_uvfull` indicates the entire matrix is stored.]

```

svd(
    length_type rows,
    length_type columns,
    storage_type ustorage,
    storage_type vstorage)
    VSIP_THROW((std::bad_alloc));

```

Requires: Positive rows . Positive columns .

Effects: Constructs a `svd` object that will decompose rows by columns matrixes.

Throws: `std::bad_alloc` indicating memory allocation for the returned `lud` object failed.

Note: This functional corresponds to VSIPL functions `vsip_svd_create_f` or `vsip_csvd_create_f`.

```
length_type rows() const VSIP_NOTHROW;
```

Returns: The number of rows in a matrix to decompose.

Note: This function corresponds to VSIPL functions `vsip_svd_getattr_f` and `vsip_csvd_getattr_f`.

```
length_type columns() const VSIP_NOTHROW;
```

Returns: The number of columns in a matrix to decompose.

Note: This function corresponds to VSIPL functions `vsip_svd_getattr_f` and `vsip_csvd_getattr_f`.

```
storage_type ustorage() const VSIP_NOTHROW;
```

Returns: How the decomposition matrix U should be stored by this object, if at all.

Note: This function corresponds to VSIPL functions `vsip_svd_getattr_f` and `vsip_csvd_getattr_f`.

```
storage_type vstorage() const VSIP_NOTHROW;
```

Returns: How the decomposition matrix V^T or V^H should be stored by this object, if at all.

Note: This function corresponds to VSIPL functions `vsip_svd_getattr_f` and `vsip_csvd_getattr_f`.

```
template <typename Block>  
const_Vector<scalar_f, unspecified>  
decompose(Matrix<T, Block> m)  
VSIP_THROW((std::bad_alloc, computation_error));
```

Requires: `m.size(0) == this->rows()` . `m.size(1) == this->columns()` . `ReturnMechanism == by_value` . `m` must be modifiable.

Effects: Performs a singular-value decomposition of m . If T is not a specialization of `complex`, $m = USV^T$, where square orthogonal matrix U has the same number of rows as m , S is a matrix with the same shape as m 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 m . If T is a specialization of `complex`, $m = USV^H$, where U , S , and V are similar to those described above except U and V are unitary, not orthogonal, matrices. If `ustorage() == svd_uvnos`, U is not stored. If `ustorage() == svd_uvpart`, the first p columns of U are stored. If `ustorage() == svd_uvfull`, all columns are stored. V^T or V^H , depending on whether T is not or is a specialization of `complex`, respectively, is similarly stored.

Returns: A `const_Vector` with `length_type` p containing the singular values of m in nonincreasing order.

Postconditions: The matrix m may not be modified as long as its decomposition may be used.

Throws: `std::bad_alloc` upon memory allocation, and `computation_error` if the decomposition could not be computed.

Note: Memory may be allocated; the object's memory requirements are not specified. This function corresponds to the functionality of VSIPL functions `vsip_svd_f` or `vsip_csvd_f`. The returned vector's block's type does not necessarily equal `Block`.

```
template <typename Block0,
          typename Block1>
bool
decompose (
    Matrix<T, Block0>      m,
    Vector<scalar_f, Block1> destination)
VSIP_NOTHROW;
```

Requires: `m.size(0) == this->rows()` . `m.size(1) == this->columns()` . `destination.size() == p`. m and `destination` must be modifiable. m and `destination` must not overlap. `ReturnMechanism == by_reference`.

Effects: Performs a singular-value decomposition of m . The singular values of m are stored in `destination`. If T is not a specialization of `complex`, $m = USV^T$, where square orthogonal matrix U has the same number of rows as m , S is a matrix with the same shape as m 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 m . If T is a specialization of `complex`, $m = USV^H$, where U , S , and V are similar to those described above except U and V are unitary, not orthogonal, matrices. If `ustorage() == svd_uvsnos`, U is not stored. If `ustorage() == svd_uvpart`, the first p columns of U are stored. If `ustorage() == svd_uvfull`, all columns are stored. V^T or V^H , depending on whether T is not or is a specialization of `complex`, respectively, is similarly stored.

Returns: `true` if the decomposition succeeds.

Postconditions: The matrix `m` may not be modified as long as its decomposition may be used.

Note: This function corresponds to the functionality of VSIPL functions `vsip_svd_f` or `vsip_csvd_f`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename         Block>
const_Matrix<T, unspecified>
produ(
    const_Matrix<T, Block> m)
const VSIP_THROW((computation_error));
```

Requires: `ReturnMechanism == by-value`. A call to decompose must have occurred for this object with `this->ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. The number of rows and columns of `m` depends on the values of `tr`, `ps`, and `this->ustorage()`. For `this->ustorage() == svd_uvpart`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	p, s	$s, \text{rows}()$
<code>tr == mat_trans</code>	$\text{rows}(), s$	s, p
<code>tr == mat_herm</code>	$\text{rows}(), s$	s, p

where s is an arbitrary positive `length_type`. For `this->ustorage() == svd_uvfull`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	$\text{rows}(), s$	$s, \text{rows}()$
<code>tr == mat_trans</code>	$\text{rows}(), s$	$s, \text{rows}()$
<code>tr == mat_herm</code>	$\text{rows}(), s$	$s, \text{rows}()$

Returns: The product of U and m . The actual product and its number of rows and columns depends on the values of `tr`, `ps`, and `this->ustorage()` and whether `T` is not or is a specialization of `complex`. For `this->ustorage() == svd_uvpart`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	$U_{m, \text{rows}(), s}$	mU, s, p
<code>tr == mat_trans, T</code>	$U^T_{m, p, s}$	$mU^T, s, \text{rows}()$
<code>tr == mat_herm, complex<T></code>	$U^H_{m, p, s}$	$mU^H, s, \text{rows}()$

where s is the same variable as above. For `this->ustorage() == svd_uvfull`,

	<code>ps == mat_lside</code>	<code>ps == mat_rside</code>
<code>tr == mat_ntrans</code>	$U_{m, \text{rows}(), s}$	$mU, s, \text{rows}()$
<code>tr == mat_trans, T</code>	$U^T_{m, \text{rows}(), s}$	$mU^T, s, \text{rows}()$
<code>tr == mat_herm, complex<T></code>	$U^H_{m, \text{rows}(), s}$	$mU^H, s, \text{rows}()$

Throws: `computation_error` if the product fails.

Note: This function corresponds to VSIPL functions `vsip_svdprodu_f` and `vsip_csvdprodu_f`. The returned matrix's block's type does not necessarily equal `Block`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename         Block0,
         typename         Block1>
bool
produ(
    const_Matrix<T, Block0> m,
    Matrix<T, Block1>      destination)
const VSIP_NOTHROW;
```

Requires: `ReturnMechanism == by-reference.m` and `destination` must not overlap. A call to decompose must have occurred for this object with `this->ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. The number of rows and columns of m depends on the values of `tr`, `ps`, and `this->ustorage()`. For `ustorage() == svd_uvpart`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	p, s	$s, \text{rows}()$
tr == mat_trans	$\text{rows}(), s$	s, p
tr == mat_herm	$\text{rows}(), s$	s, p

where s is an arbitrary positive length_type . For this->ustorage() == svd_uvfull,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$\text{rows}(), s$	$s, \text{rows}()$
tr == mat_trans	$\text{rows}(), s$	$s, \text{rows}()$
tr == mat_herm	$\text{rows}(), s$	$s, \text{rows}()$.

The required number of rows and columns of destination depends on the values of tr, ps, and this->ustorage() . For this->ustorage() == svd_uvpart,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$\text{rows}(), s$	s, p
tr == mat_trans	p, s	$s, \text{rows}()$
tr == mat_herm	p, s	$s, \text{rows}()$

where s is the same variable as above. For this->ustorage() == svd_uvfull,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$\text{rows}(), s$	$s, \text{rows}()$
tr == mat_trans	$\text{rows}(), s$	$s, \text{rows}()$
tr == mat_herm	$\text{rows}(), s$	$s, \text{rows}()$

Effects: Stores the product of U and m in destination . The actual product depends on the values of tr and ps and whether T is not or is a specialization of complex:

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	U_m	mU
tr == mat_trans, T	$U^T m$	mU^T
tr == mat_herm, complex<T>	$U^H m$	mU^H

Returns: true if the product succeeds.

Note: This function corresponds to VSIPL functions `vsip_svdprodu_f` and `vsip_csvdprodu_f`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename        Block>
const_Matrix<T, unspecified>
prodv(const_Matrix<T, Block> m)
const VSIP_THROW((computation_error));
```

Requires: ReturnMechanism == `by_value`. A call to decompose must have occurred for this object with `this->vstorage()` equaling either `svd_uvpart` or `svd_uvfull`. The number of rows and columns of `m` depends on the values of `tr`, `ps`, and `this->vstorage()`. For `this->vstorage() == svd_uvpart`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	p, s	$s, \text{columns}()$
tr == mat_trans	$\text{columns}(), s$	s, p
tr == mat_herm	$\text{columns}(), s$	s, p

where s is an arbitrary positive `length_type`. For `this->vstorage() == svd_uvfull`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$\text{columns}(), s$	$s, \text{columns}()$
tr == mat_trans	$\text{columns}(), s$	$s, \text{columns}()$
tr == mat_herm	$\text{columns}(), s$	$s, \text{columns}()$.

Returns: The product of V and m . The actual product and its number of rows and columns depends on the values of `tr`, `T`, `ps`, and `this->vstorage()`. For `this->vstorage() == svd_uvpart`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$Vm, \text{columns}(), s$	mV, s, p
tr == mat_trans, T	$V^T m, p, s$	$mV^T, s, \text{columns}()$
tr == mat_herm, complex<T>	$V^H m, p, s$	$mV^H, s, \text{columns}()$

where s is the same variable as above. For `this->vstorage() == svd_uvfull`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$V_{m, \text{columns}(), s}$	$mV, s, \text{columns}()$
tr == mat_trans, T	$V^T m, \text{columns}(), s$	$mV^T, s, \text{columns}()$
tr == mat_herm, complex<T>	$V^H m, \text{columns}(), s$	$mV^H, s, \text{columns}()$

Throws: `computation_error` if the product fails.

Note: This function corresponds to VSIPL functions `vsip_svdprodv_f` and `vsip_csvdprodv_f`. The returned matrix's block's type does not necessarily equal `Block`.

```
template <mat_op_type      tr,
         product_side_type ps,
         typename        Block0,
         typename        Block1>
bool
prodv(
    const_Matrix<T, Block0> m,
    Matrix<T, Block1>      destination)
const VSIP_NOTHROW;
```

Requires: `ReturnMechanism == by_reference.m` and `destination` must not overlap. A call to decompose must have occurred for this object with `vstorage()` equaling either `svd_uvpart` or `svd_uvfull`. The number of rows and columns of `m` depends on the values of `tr`, `ps`, and `vstorage()`. For `vstorage() == svd_uvpart`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	p, s	$s, \text{columns}()$
tr == mat_trans	$\text{columns}(), s$	s, p
tr == mat_herm	$\text{columns}(), s$	s, p

where s is an arbitrary positive `length_type`. For `vstorage() == svd_uvfull`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	$\text{columns}(), s$	$s, \text{columns}()$
tr == mat_trans	$\text{columns}(), s$	$s, \text{columns}()$
tr == mat_herm	$\text{columns}(), s$	$s, \text{columns}()$.

The number of rows and columns of `destination` depends on the values of `tr`, `ps`, and `vstorage()`. For `vstorage() == svd_uvpart`,

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	columns(), s	s, p
tr == mat_trans	p, s	s, columns()
tr == mat_herm	p, s	s, columns()

where *s* is the same variable as above. 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()

Effects: Stores the product of *V* and *m* in `destination`. The actual product and its number of rows and columns depends on the values of `tr` and `ps` and whether *T* is not or is a specialization of `complex`.

	ps == mat_lside	ps == mat_rside
tr == mat_ntrans	V_m	mV
tr == mat_trans, T	V_m^T	mV^T
tr == mat_herm, complex<T>	V_m^H	mV^H

Returns: true if the product succeeds.

Note: This function corresponds to VSIPL functions `vsip_svdprodv_f` and `vsip_csvdprodv_f`.

```
const_Matrix<T, unspecified>
u(
    index_type low,
    index_type high)
const VSIP_THROW((computation_error));
```

Requires: A call to decompose must have occurred for this object with `ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. $0 \leq \text{low} \leq \text{high}$. If `this->ustorage() == svd_uvpart`, $\text{high} \leq p$. If `this->ustorage() == svd_uvfull`, $\text{high} \leq \text{this->rows()}$. `ReturnMechanism == by-value`.

Returns: A submatrix of *U* containing columns `low`, `low+1`, ..., `high`, inclusive.

Throws: `computation_error` if the most recent decomposition failed.

Note: This function corresponds to VSIPL functions `vsip_svdmatu_f` and `vsip_csvdmatu_f`.

```
template <typename Block>
bool
u(
    index_type      low,
    index_type      high,
    Matrix<T, Block> destination)
const VSIP_NOTHROW;
```

Requires: A call to decompose must have occurred for this object with `this->ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. $0 \leq \text{low} \leq \text{high}$. If `this->ustorage() == svd_uvpart`, $\text{high} \leq p$. If `this->ustorage() == svd_uvfull`, $\text{high} \leq \text{this->rows() . destination.size(0) == this->rows() . destination.size(1) == \text{high} - \text{low} + 1$. `destination` must be modifiable. `ReturnMechanism == by_reference`.

Effects: Store a submatrix of U containing columns `low`, `low+1`, ..., `high`, inclusive, into `destination`.

Returns: `true` if a matrix is stored.

Note: This function corresponds to VSIPL functions `vsip_svdmatu_f` and `vsip_csvdmatu_f`.

```
Matrix<T, unspecified>
v(
    index_type low,
    index_type high)
const VSIP_THROW((computation_error));
```

Requires: A call to decompose must have occurred for this object with `this->vstorage()` equaling either `svd_uvpart` or `svd_uvfull`. $0 \leq \text{low} \leq \text{high}$. If `this->vstorage() == svd_uvpart`, $\text{high} \leq p$. If `this->vstorage() == svd_uvfull`, $\text{high} \leq \text{this->columns() .$

Returns: A submatrix of V containing columns `low`, `low+1`, ..., `high`, inclusive.

Throws: `computation_error` if the most recent decomposition failed.

Note: This function corresponds to VSIPL functions `vsip_svdmatv_f` and `vsip_csvdmatv_f`.

```
template <typename Block>
bool
v(
    index_type      low,
    index_type      high,
    Matrix<T, Block> destination)
const VSIP_NOTHROW;
```

Requires: A call to decompose must have occurred for this object with `this->vstorage()` equaling either `svd_uvpart` or `svd_uvfull.0` `<= low <= high` . If `this->vstorage() == svd_uvpart`, `high <= p`. If `this->vstorage() == svd_uvfull`, `high <= this->columns()` . `destination.size(0) == this->columns()` . `destination.size(1) == high - low + 1` . `destination` must be modifiable. `ReturnMechanism == by_reference` .

Effects: Store a submatrix of V containing columns `low`, `low+1`, ..., `high`, inclusive, into `destination` .

Returns: `true` if a matrix is stored.

Note: This function corresponds to VSIPL functions `vsip_svdmatv_f` and `vsip_csvdmatv_f`.

- 1 This clause specifies functions that select view indices or values satisfying a criterion. It also specifies functions that generate views from scalars and specifies functions manipulating views.

Header <vsip/selgen.hpp> synopsis

```
namespace vsip {
    // selection functions

    // first
    template <typename FnObject,
              typename T1, typename T2,
              typename Block1, typename Block2>
    index_type first(index_t, FnObject,
                    const_Vector<T1, Block1>,
                    const_Vector<T2, Block2>);

    template <template <typename, typename> class const_View,
              typename T,
              typename Block1, typename Block2>
    length_type indexbool(
        const_View<T, Block1> source,
        Vector<Index<const_View<T, Block>::dim>, Block2> indices)
    VSIP_NOTHROW;

    // gather
    template <template <typename, typename> class const_View,
              typename T,
              typename Block0, typename Block1>
    const_Vector<T, unspecified>
    gather(const_View<T, Block0>,
          const_Vector<Index<const_View<T, Block>::dim>,
                    Block1>)
    VSIP_NOTHROW;
}
```

```

// scatter
template <template <typename, typename> class const_View,
         typename T, typename Block0, typename Block1,
         typename Block2>
void
scatter(const_Vector<T, Block0>,
        const_Vector<Index<const_View<T,Block2>::dim>,
        Block1>,
        typename
        ViewConversion<const_View, T, Block2>::view_t)
VSIP_NOTHROW;

// ramp
template <typename T>
const_Vector<T, unspecified> ramp(T a, T b, length_type len)
VSIP_NOTHROW;

// clipping and inverse clipping
template <typename Tout, typename Tin0, typename Tin1,
         template <typename, typename> class const_View,
         typename Block>
const_View<Tout, unspecified>
clip(const_View<Tin0, Block>,
     Tin1 lower_threshold,
     Tin1 upper_threshold,
     Tout lower_clip_value,
     Tout upper_clip_value)
VSIP_NOTHROW;

template <typename Tout, typename Tin0, typename Tin1,
         template <typename, typename> class const_View,
         typename Block>
const_View<Tout, unspecified>
invclip(const_View<Tin0, Block>,
        Tin1 lower_threshold,
        Tin1 middle_threshold,
        Tin1 upper_threshold,
        Tout lower_clip_value,
        Tout upper_clip_value)
VSIP_NOTHROW;

// manipulation functions
template <typename T0, typename T1,
         template <typename, typename> class View,
         typename Block0, typename Block1>

```

```

void
swap(View<T0, Block0>, View<T1, Block1>) VSIP_NOTHROW;
}

```

9.1 Selection functions

[selgen.selection]

- 1 A *selection function* finds one or more values or indices satisfying a given criterion in one or more views.

9.1.1 Find leftmost true pair

[selgen.selection.first]

```

template <typename FnObject,
          typename T1,
          typename T2,
          typename Block1,
          typename Block2>
index_type
first(
    index_type          j,
    FnObject            obj,
    const_Vector<T1, Block1> v,
    const_Vector<T2, Block2> w);

```

Requires: The only specializations which must be supported have `FnObject` the same as `bool (*)(scalar_f, scalar_f)`, `bool (*)(scalar_i, scalar_i)`, `bool (*)(bool, bool)`, `bool (*)(Index<1> const&, Index<1> const&)`, `bool (*)(Index<2> const&, Index<2> const&)`, and `bool (*)(Index<3> const&, Index<3> const&)`. An implementation is permitted to prevent instantiation for other choices for `FnObject`. If `t1` is an object of type `T1` and `t2` is an object of type `T2`, `obj(t1, t2)` must be valid. `v` and `w` must be element conformant.

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()`.

Note: This function corresponds to the VSIPL functions `vsip_vfirst_f`, `vsip_vfirst_i`, `vsip_vfirst_bl`, `vsip_vfirst_vi`, `vsip_vfirst_mi`, and `vsip_vfirst_ti`.

9.1.2 Find indices of non-false values

[selgen.selection.indexbool]

```

template <template <typename, typename> class const_View,
        typename                               T,
        typename                               Block1,
        typename                               Block2>
length_type
indexbool (
    const_View<T, Block1>                       source,
    Vector<Index<const_View<T, Block1>::dim>, Block2> indices)
VSIP_NOTHROW;

```

Requires: The only specializations which must be supported have `const_View` the same as `const_Vector` or `const_Matrix` and `T` the same as `bool`. An implementation is permitted to prevent instantiation for other choices for `const_View` and `T`. `Block2` must be modifiable. `indices.size()` must be at least equal to the number of non-false values in `source`.

Effects: Let `len` be the value returned by the function. If `len == 0`, then `indices` is not modified. Otherwise, `indices.get(0)`, ..., `indices.get(len-1)` contain distinct `Index<const_View<T, Block1>::dim>` values `v` such that the source value at `v` is not false. Furthermore, values in `indices` are lexicographically increasingly ordered.

Returns: The number of non-false values in `source`.

Note: This function corresponds to the VSIP functions `vsip_vindexbool_b1` and `vsip_mindexbool_b1` but does not modify the given `Vector`'s size.

9.1.3 Gathering specified values

[selgen.selection.gather]

- 1 A `gather` instantiation returns a `const_Vector` with view values specified by a `const_Vector` of indices.

```

template <template <typename, typename> class const_View,
        typename                               T,
        typename                               Block0,
        typename                               Block1>
const_Vector<T, unspecified>
gather (
    const_View<T, Block0>                       v,

```

```
const_Vector<Index<View<T, Block0>::dim>, Block1> idx)
VSIP_NOTHROW;
```

Requires: An implementation must support specializations with `const_View` the same as `const_Vector` or `const_Matrix` and `T` the same as `scalar_i`, `scalar_f`, or `cscalar_f`. It must also support specializations with `const_View` the same as `const_Tensor` and `T` the same as `scalar_i`, `scalar_f`, `cscalar_f`, `cscalar_i`, or `bool`. An implementation is permitted to prevent instantiation for other choices for `const_View` and `T`. For all $0 \leq i < \text{idx.size}()$, `idx.get(i)` must be in `v`'s domain.

Returns: A `const_Vector` `w` having `w.size() == idx.size()`. For all indices $0 \leq i < \text{idx.size}()$, `w.get(i) == v.get(idx.get(i))`.

Note: This function corresponds to the VSIPL functions `vsip_vgather_i`, `vsip_mgather_i`, `vsip_tgather_i`, `vsip_vgather_f`, `vsip_mgather_f`, `vsip_tgather_f`, `vsip_cvgather_f`, `vsip_cmgather_f`, `vsip_ctgather_f`, `vsip_ctgather_i`, and `vsip_tgather_bl`. The returned vector's block is not necessarily the same as `Block0` or `Block1`.

9.1.4 Scattering specified values

[selgen.selection.scatter]

- 1 [Note: The composition of a scatter instantiation and its analogous gather instantiation yields the identity function.]

```
template <template <typename, typename> class const_View,
        typename T,
        typename Block0,
        typename Block1,
        typename Block2>
void
scatter (
    const_Vector<T, Block0> v,
    const_Vector<Index<View<T, Block2>::dim>, Block1> idx,
    typename ViewConversion<const_View, T, Block2>::view_type out)
VSIP_NOTHROW;
```

Requires: An implementation must support specializations with `const_View` the same as `const_Vector` or `const_Matrix` and `T` the same as

`scalar_i`, `scalar_f`, or `cscalar_f`. It must also support specializations with `const_View` the same as `const_Tensor` and `T` the same as `scalar_i`, `scalar_f`, `cscalar_f`, `cscalar_i`, or `bool`. An implementation is permitted to prevent instantiation for other choices for `const_View` and `T`. `v` and `idx` must be element conformant. All `idx` entries must be within the domain of `out.block()`. `v` must not overlap with `out`. `idx` must not overlap with `out`.

Effects: For all indices $0 \leq i < v.\text{domain.size}()$, `out.get(idx.get(i)) == v.get(i)` unless `idx` contains duplicate entries in which case the stored value is undefined.

Note: This function corresponds to the VSIPL functions `vsip_vscatter_i`, `vsip_mscatter_i`, `vsip_tscatter_i`, `vsip_vscatter_f`, `vsip_mscatter_f`, `vsip_tscatter_f`, `vsip_cvscatter_f`, `vsip_cmscatter_f`, `vsip_ctscatter_f`, `vsip_ctscatter_i`, and `vsip_tscatter_bl`.

9.2 Generation functions

[selgen.generate]

- 1 A *generation function* generates a view from its given parameters, which usually include scalars and/or views.
- 2 [*Note:* Unlike most VSIPL++ functions, these functions modify their view arguments. Most are obviated by scalar assignment syntax, which may be faster. For example, the VSIPL fill functions, `vsip_vfill_i`, `vsip_vfill_f`, `vsip_cvfill_f`, `vsip_mfill_i`, `vsip_mfill_f`, and `vsip_cmfill_f`, are obviated by scalar assignment.]

9.2.1 Filling with a linear function

[selgen.generate.ramp]

```
template <typename T>
const_Vector<T, unspecified>
ramp(
    T          a,
    T          b,
    length_type len)
VSIP_NOTHROW;
```

Requires: `len > 0`. The only specializations which must be supported have `T` the same as `scalar_i` or `scalar_f`. An implementation is permitted to prevent instantiation for other choices.

Returns: A `const_Vector w` having `w.size() == len`. For $0 \leq i < \text{len}$, `w.get(i) == a + i * b`.

Note: This function corresponds to the VSIPL functions `vsip_vramp_f` and `vsip_vramp_i`.

9.3 Clipping and inverse clipping

[selgen.clip]

```
template <typename Tout,
          typename Tin0,
          typename Tin1,
          template <typename, typename> class const_View,
          typename Block>
const_View<Tout, unspecified>
clip(
    const_View<Tin0, Block>,
    Tin1 lower_threshold,
    Tin1 upper_threshold,
    Tout lower_clip_value,
    Tout upper_clip_value)
VSIP_NOTHROW;
```

Requires: `Tin0` and `Tin1` must be comparable. The only specializations which must be supported have `Tout`, `Tin0`, and `Tin1` all `scalar_f` or all `scalar_i` and `const_View` either `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiations for other values of `Tout`, `Tin0`, `Tin1`, and `View`.

Returns: The element-wise extension of the one-element clip function. The one-element clip function operating on a value `v` returns a value determined by applying these rules sequentially until a condition is satisfied:

1. If $v \leq \text{lower_threshold}$, return `lower_clip_value`.
2. If $v < \text{upper_threshold}$, return `v`.
3. Return `upper_clip_value`.

Note: This function provides the functionality of the VSIPL functions `vsip_vclip_f`, `vsip_vclip_i`, `vsip_mclip_f`, and `vsip_mclip_i`. The returned view's block is not necessarily the same as `Block`.

```
template <typename Tout,
          typename Tin0,
```

```
        typename                               Tin1,
        template <typename, typename> class const_View,
        typename                               Block>
const_View<Tout, unspecified>
invclip(
    const_View<Tin0, Block>,
    Tin1 lower_threshold,
    Tin1 middle_threshold,
    Tin1 upper_threshold,
    Tout lower_clip_value,
    Tout upper_clip_value)
VSIP_NOTHROW;
```

Requires: Tin0 and Tin1 must be comparable. The only specializations which must be supported have Tout, Tin0, and Tin1 all scalar_*f* or all scalar_*i* and const_View either const_Vector or const_Matrix. An implementation is permitted to prevent instantiations for other values of Tout, Tin0, Tin1, and View.

Returns: The element-wise extension of the one-element inverse clip function. The one-element inverse clip function operating on a value *v* returns a value determined by applying these rules sequentially until a condition is satisfied:

1. If $v < \text{lower_threshold}$, return v .
2. If $v < \text{middle_threshold}$, return lower_clip_value .
3. If $v \leq \text{upper_threshold}$, return upper_clip_value .
4. Otherwise, return v .

Note: This function provides the functionality of the VSIPL functions `vsip_vinvclip_f`, `vsip_vinvclip_i`, `vsip_minvclip_f`, and `vsip_minvclip_i`.

9.4 Manipulation functions

[selgen.manipulation]

- 1 A *manipulation function* modifies values in one or more views.

```
template <typename                               T0,
        typename                               T1,
        template <typename, typename> class View,
        typename                               Block0,
        typename                               Block1>
```

```
void  
swap(  
    View<T0, Block0> v,  
    View<T1, Block1> w)  
VSIP_NOTHROW;
```

Requires: The only specializations which must be supported are for `T0` and `T1` both the same as `scalar_i`, `scalar_f`, or `cscalar_f` and `View` the same as `Vector` or `Matrix`. An implementation is permitted to prevent instantiations for other values of `T0`, `T1`, and `View`. `Block0` and `Block1` must be modifiable. `v` and `w` must be element-conformant.

Effects: For $0 \leq i < v.size()$, the values `v.get(i)` and `w.get(i)` are swapped, i.e., exchanged.

Note: This function provides part of the functionality of the VSIP functions `vsip_vswap_i`, `vsip_vswap_f`, `vsip_cvswap_f`, `vsip_mswap_i`, `vsip_mswap_f`, and `vsip_cmswap_f`.

- 1 This clause specifies random number generation.

10.1 Rand

[random.rand]

Header `<vsip/random.hpp>` synopsis

```

namespace vsip {
    template <typename T = VSIP_DEFAULT_VALUE_TYPE>
    class Rand {
    public:
        // view types
        typedef const_Vector<T, unspecified> vector_type;
        typedef const_Matrix<T, unspecified> matrix_type;

        // constructors, copies, assignments, and destructors
        Rand(index_type, bool = true) VSIP_THROW((std::bad_alloc));
        Rand(index_type, index_type, index_type, bool = true)
            VSIP_THROW((std::bad_alloc));
    private:
        Rand(Rand const&) VSIP_NOTHROW;
        Rand& operator=(Rand const&) VSIP_NOTHROW;
    public:
        ~Rand() VSIP_NOTHROW;

        // number generators
        T randu() VSIP_NOTHROW;
        T randn() VSIP_NOTHROW;
        vector_type<T, unspecified> randu(length_type) VSIP_NOTHROW;
        matrix_type<T, unspecified> randu(length_type, length_type)
            VSIP_NOTHROW;
        vector_type<T, unspecified> randn(length_type) VSIP_NOTHROW;
        matrix_type<T, unspecified> randn(length_type, length_type)
            VSIP_NOTHROW;
    };
}

```

```
};  
}
```

- 1 [Note: The VSIPL++ `Rand<T>` template is a generalization of the VSIPL `rand` object.]
- 2 The only specializations of `Rand` which must be supported are `Rand<scalar_f>` and `Rand<cscalar_f>`. An implementation is permitted to prevent instantiation of `Rand<T>` for other choices of `T`.

10.1.1 View types

[**random.rand.view.types**]

- 1 `vector_type` specifies the type of a `const_Vector` with value type `T` and an unspecified block type.
- 2 `matrix_type` specifies the type of a `const_Matrix` with value type `T` and an unspecified block type.

10.1.2 Constructors, copy, assignment, and destructor

[**random.rand.-constructors**]

```
Rand(index_type seed, bool portable = true)  
VSIP_THROW((std::bad_alloc));
```

Effects: Constructs a random number generator object using the specified `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.”

Throws: `std::bad_alloc` upon memory allocation error.

Note: This constructor facilitates computation using one processor. Use the four-parameter constructor for multiprocessor execution.

```
Rand(  
    index_type seed,  
    index_type numprocs,  
    index_type id,  
    bool        portable = true)  
VSIP_THROW((std::bad_alloc));
```

Requires: $0 < \text{id} \leq \text{numprocs} \leq 2^{31} - 1$.

Effects: Constructs a random number generator object. If `portable == false`, the random number generator characteristics are implementation defined. Otherwise, the random number generator object obeys the VSIPL 1.1 API sections listed above, the seed sequence is split into `numprocs` equal-length subsequences, and the object's initial seed equals `seed + (id - 1) * l`, where l is the subsequence length.

Throws: `std::bad_alloc` upon memory allocation error.

Note: `numprocs` indicates the total number of processors, and `id` indicates a processor ID.

10.1.3 Number generators

[random.rand.generate]

T

```
randu() VSIP_NOTHROW;
```

Returns: A uniformly distributed random deviate over the open interval $(0, 1)$. A complex random number has real and imaginary components where each component is uniformly distributed over $(0, 1)$.

vector_type

```
randu(length_type len) VSIP_NOTHROW;
```

Requires: `len > 0`.

Returns: A `vector_type` `v` containing `len` uniformly distributed pseudo-random numbers from the open interval $(0, 1)$.

Postconditions: `v.length() == len`.

matrix_type

```
randu(length_type rows, length_type columns) VSIP_NOTHROW;
```

Requires: `rows > 0 && columns > 0`.

Returns: A `matrix_type` `m` containing `rows*columns` uniformly distributed pseudo-random numbers from the open interval $(0, 1)$.

Postconditions: `m.size(0) == rows`, `m.size(1) == columns`.

T

```
randn() VSIP_NOTHROW;
```

Returns: A normally distributed random deviate having mean zero and unit variance, i.e., $N(0, 1)$.

Note: For a mathematical description how to generate pseudo-random normally distributed deviates using `randu`, see the VSIP description of `vsip_randn_f` and `vsip_crandn_f`.

`vector_type`

```
randn(length_type len) VSIP_NOTHROW;
```

Requires: `len > 0`.

Returns: A `vector_type` `v` containing `len` normally distributed pseudo-random numbers from the distribution $N(0, 1)$.

Postconditions: `v.length() == len`.

Note: For a mathematical description how to generate pseudo-random normally distributed deviates using `randu`, see the VSIP description of `vsip_randn_f` and `vsip_crandn_f`.

`matrix_type`

```
randn(length_type rows, length_type columns) VSIP_NOTHROW;
```

Requires: `row > 0 && columns > 0`.

Returns: A `matrix_type` `m` containing `rows * columns` normally distributed pseudo-random numbers from the distribution $N(0, 1)$.

Postconditions: `m.size(0) == rows.m.size(1) == columns`.

Note: For a mathematical description how to generate pseudo-random normally distributed deviates using `randu`, see the VSIP description of `vsip_randn_f` and `vsip_crandn_f`.

- 1 This clause specifies classes and functions for signal processing.
- 2 [*Note:* The VSIP 1.1 API Signal Processing introduction contains references to signal processing books and articles.]

Header <vsip/signal.hpp> synopsis

```

namespace vsip {
  // implementation hints
  enum alg_hint_type { alg_time, alg_space, alg_noise };

  // fast Fourier transforms
  int const fft_fwd = unspecified;
  int const fft_inv = unspecified;

  template <template <typename, typename> class const_View,
            typename      InputType,
            typename      OutputType,
            int            SD = 0,
            return_mechanism_type ReturnMechanism = by_value,
            unsigned       number_of_times = 0,
            alg_hint_type  ah = alg_time>
  class Fft;

  template <typename      InputType,
            typename      OutputType,
            int            SD = row,
            int            direction = fft_fwd,
            return_mechanism_type ReturnMechanism = by_value,
            unsigned       number_of_times = 0,
            alg_hint_type  ah = alg_time>
  class Fftm;

  // convolutions and correlations
  enum support_region_type { support_full, support_same, support_min };

```

11. [SIGNAL] SIGNAL PROCESSING

```
enum symmetry_type { nonsym, sym_even_len_odd, sym_even_len_even };

template <template <typename, typename> class const_View,
        symmetry_type      symm,
        support_region_type supp,
        typename           T = VSIP_DEFAULT_VALUE_TYPE,
        unsigned           number_of_times = 0,
        alg_hint_type      ah = alg_time>
class Convolution;

template <template <typename, typename> class const_View,
        support_region_type supp,
        typename           T = VSIP_DEFAULT_VALUE_TYPE,
        unsigned           number_of_times = 0,
        alg_hint_type      ah = alg_time>
class Correlation;

// window creation functions
const_Vector<scalar_f, unspecified>
blackman(length_type) VSIP_THROW((std::bad_alloc));

const_Vector<scalar_f, unspecified>
cheby(length_type, scalar_f)
    VSIP_THROW((std::bad_alloc));

const_Vector<scalar_f, unspecified>
hanning(length_type) VSIP_THROW((std::bad_alloc));

const_Vector<scalar_f, unspecified>
kaiser(length_type, scalar_f) VSIP_NOTHROW;

// Fir and Iir filters
enum obj_state { state_no_save, state_save };

template <typename      T = VSIP_DEFAULT_VALUE_TYPE,
        symmetry_type symm = nonsym,
        obj_state     contin_filtering = state_save,
        unsigned      number_of_times = 0,
        alg_hint_type ah = alg_time>
class Fir;

template <typename      T = VSIP_DEFAULT_VALUE_TYPE,
        obj_state     contin_filtering = state_save,
        unsigned      number_of_times = 0,
        alg_hint_type ah = alg_time>
```

```

class Iir;

// histogram and frequency swap functions
template <template <typename, typename> class const_View,
        typename
        T>
class Histogram;

template <template <typename, typename> class const_View,
        typename
        T,
        typename
        Block>
const_View<T, unspecified>
freqswap(const_View<T,Block>) VSIP_NOTHROW;
}

```

11.1 Implementation hints

[signal.hint]

- 1 An implementation may use `alg_hint_type` or an integer to indicate how to optimize its computation or resource use. The VSIPL 1.1 API for an implementation's use of the hints are incorporated by reference.

```

namespace vsip {
    enum alg_hint_type { alg_time, alg_space, alg_noise };
}

```

[*Note:* The VSIPL 1.1 API specifies an unsigned integer's value indicates the probable number of uses, with a value of zero indicating semi-infinity, i.e., many times. The `alg_hint_type` value indicates whether total execution time, total execution memory, or numeric noise should be minimized.]

11.2 Single fast Fourier transformations

[signal.fft]

- 1 Applying an `Fft` object on a view performs a single fast Fourier transform on the entire view. [*Note:* Multiple fast Fourier transforms are specified in [signal.fftm].]
- 2 All VSIPL 1.1 API complexity requirements for FFTs are incorporated by reference.
- 3 [*Note:* For mathematical descriptions of FFTs, see the VSIPL description of `vsip_ccffftop_f`, `vsip_crffftop_f`, `vsip_rcffftop_f`, `vsip_ccffftop_create_f`, `vsip_ccffft2dop_f`, `vsip_crffft2dop_f`, `vsip_rcffft2dop_f`, `vsip_ccffft2dop_create_f`,

`vsip_ccfft3dop_f`, `vsip_crfft3dop_f`, `vsip_rcfft3dop_f`, and `vsip_ccfft3dop_create_f`.]

- 4 For transformations of the entire view, `Fft` supports different computations dependent on the input element type, output element type, a specified direction or a special dimension (“sd”), and the dimensionalities of the input and output views. They must satisfy one of these criteria:

input type / output type	sd	input size	output size
<code>complex<T>/complex<T></code>	<code>fft_fwd</code>	M	M
		$M \times N$	$M \times N$
		$M \times N \times P$	$M \times N \times P$
	<code>fft_inv</code>	M	M
		$M \times N$	$M \times N$
		$M \times N \times P$	$M \times N \times P$
<code>T/complex<T></code>	0	M	$(M/2 + 1)$
		$M \times N$	$(M/2 + 1) \times N$
		$M \times N \times P$	$(M/2 + 1) \times N \times P$
	1	$M \times N$	$M \times (N/2 + 1)$
		$M \times N \times P$	$M \times (N/2 + 1) \times P$
	2	$M \times N \times P$	$M \times N \times (P/2 + 1)$
<code>complex<T>/T</code>	0	$(M/2 + 1)$	M
		$(M/2 + 1) \times N$	$M \times N$
		$(M/2 + 1) \times N \times P$	$M \times N \times P$
	1	$M \times (N/2 + 1)$	$M \times N$
		$M \times (N/2 + 1) \times P$	$M \times N \times P$
	2	$M \times N \times (P/2 + 1)$	$M \times N \times P$

M , N , and P are `length_types` indicating the number of rows, columns, and depth, respectively. [Note: To simplify the table, a row does not repeat a column entry if it is the same as in the previous row.]

[Note: If the input and output types are both complex, then the specific direction indicates whether a `fft_fwd` (forward) transform using positive exponentials or `fft_inv` (inverse) transform using negative exponentials should occur.]

Some criteria have input and output views whose corresponding dimensions differ in size. The special dimension (“sd”) column indicates this dimension. For such a dimension, the larger dimension size Q must be even so $Q/2 + 1$ is integral. `T` must be a type such that `complex<T>` can be instantiated.

[Example: An `Fft` using `Vectors` of `cscalar_fs` for both input and output requires the input and output `Vectors` to have the same size M . An `Fft` using `Vectors` with input type `scalar_f` and output type `cscalar_f` requires the

input Vector to have size M and the output to have size $M/2 + 1$. M must be even.]

[*Note:* The special cases for non-complex input or output type permits reducing the computation time by about a factor of two.] For the special dimension, the complex views for these cases conceptually have length N but only the first $\lfloor N/2 \rfloor + 1$ must be supplied or will be returned, as appropriate. The other values are specified by the identity $x_n = x_{N-n}^*$ for $\lfloor N/2 \rfloor < n < N$. For these complex views, the complex value at index position 0 represents the zero (DC) frequency and must have zero imaginary part. The complex value at index position $N/2$ represents the Nyquist frequency (one half the sample rate value) and must also be non-complex.

```
namespace vsip {
    int const fft_fwd = unspecified;
    int const fft_inv = unspecified;

    template <template <typename, typename> class const_View,
              typename InputType, typename OutputType,
              int SD = 0,
              return_mechanism_type ReturnMechanism = by_value,
              unsigned number_of_times = 0, alg_hint_type ah = alg_time>
    class Fft {
    public:
        // constructor, copies, assignments, and destructor
        Fft(Domain<const_View::dim> const&, scalar_f)
            VSIP_THROW((std::bad_alloc));
        Fft(Fft const&) VSIP_NOTHROW;
        Fft& operator=(Fft const&) VSIP_NOTHROW;
        ~Fft() VSIP_NOTHROW;

        // accessors
        Domain<const_View::dim> const& input_size() const
            VSIP_NOTHROW;
        Domain<const_View::dim> const& output_size() const
            VSIP_NOTHROW;
        scalar_f scale() const VSIP_NOTHROW;
        bool forward() const VSIP_NOTHROW;

        // operators
        // if ReturnMechanism is by_value:
        template <typename Block>
        const_View<OutputType, unspecified>
        operator() (const_View<InputType, Block>)
            VSIP_THROW((std::bad_alloc));

        // if ReturnMechanism is by_reference:
```

11.2.1. [SIGNAL.FFT.CONSTANTS] CONSTANTS

```
template <typename Block0, typename Block1>
const_View<OutputType, Block1>
operator() (const_View<InputType, Block0>,
           ViewConversion<View, OutputType, Block1>::view_type)
    VSIP_NOTHROW;
// if ReturnMechanism is by_reference and
// InputType == OutputType:
template <typename Block>
typename ViewConversion<View, OutputType, Block>::view_type
operator() (ViewConversion<View, OutputType, Block>::view_type)
    VSIP_NOTHROW;
};
}
```

- 5 `const_View::dim` indicates the dimensionality of `const_View`.
- 6 The template `Fft` class has seven template parameters and eighteen specializations corresponding to `[signal.fft]/4`. Specifications for the function and data members of these specializations are the same so they are presented only once below using `InputType` for the input type and `OutputType` for the output type. `View::dim` indicates the dimensionality of the view.

11.2.1 Constants

[[signal.fft.constants](#)]

- 1 The constants `fft_fwd` and `fft_inv` correspond to the forward and inverse transforms, respectively.

11.2.2 Template parameters

[[signal.fft.template](#)]

- 1 The effect of instantiating the template class `Fft` for any template `const_View` parameter other than `const_Vector`, `const_Matrix`, or `const_Tensor` is unspecified.
- 2 `InputType` and `OutputType` must obey the table above. The only specializations which must be supported have `T` equal to `scalar_f`. An implementation is permitted to prevent other instantiations. [*Note:* If `T` is `scalar_f`, `complex<T>` is `cscalar_f`.]

int SD

Requires: If `InputType` and `OutputType` differ, $0 \leq SD \ \&\& \ SD < \text{const_View::dim}$. Otherwise, `SD` must be either `fft_fwd` or `fft_inv`.

Note: If `InputType` and `OutputType` differ, its value indicates which dimension has different input and output sizes. If `InputType` and `OutputType` are the same, this indicates whether a forward or inverse transform should occur.

- 3 [Note: The `return_mechanism_type` values indicate whether operators yield values by returning them by value while others use an “output” parameter. `by_value` and `by_reference` respectively describe each.]

`unsigned number_of_times`

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

`alg_hint_type ah`

Requires: A `alg_hint_type` value.

Note: This value indicates how an implementation should optimize its computation or resource use.

11.2.3 Constructors, copy, assignment, and destructor [signal.fft.constructors]

```
Fft (
    Domain<const_View::dim> const& dom,
    scalar_f scale)
VSIP_THROW((std::bad_alloc));
```

Requires: `dom` must obey [signal.fft]/4 as determined by the template arguments.

Effects: Constructs an object of class `Fft` .

Postconditions: `this->input_size()` and `this->output_size()` correspond to the appropriate row in [signal.fft]/4. `this->scale() == scale` .

Throws: `std::bad_alloc` upon a memory allocation error.

Note: This function implements part of the functionality of the VSIPL functions `vsip_ccffftop_create_f`, `vsip_ccffftip_create_f`, `vsip_crffftop_create_f`, `vsip_rcffftop_create_f`, `vsip_ccfft2dop_create_f`, `vsip_ccfft2dip_create_f`, `vsip_crfft2dop_create_f`, `vsip_rcfft2dop_create_f`, `vsip_ccfft3dop_create_f`, `vsip_ccfft3dip_create_f`, `vsip_crfft3dop_create_f`, and `vsip_rcfft3dop_create_f`.

*11.2.4 Accessors***[signal.fft.accessors]**

```
Domain<const_View::dim> const&  
input_size() const VSIP_NOTHROW;
```

Returns: A domain object with first indices of zero, unit strides, and size equal to the input size in the appropriate row in [signal.fft]/4.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
Domain<const_View::dim> const&  
output_size() const VSIP_NOTHROW;
```

Returns: A domain object with first indices of zero, unit strides, and size equal to the output size in the appropriate row in [signal.fft]/4.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
scalar_f scale() const VSIP_NOTHROW;
```

Returns: The scalar multiple, as specified in the object's constructor.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
bool  
forward() const VSIP_NOTHROW;
```

Returns: An indication whether the transform is a forward or inverse transform.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

*11.2.5 Operators***[signal.fft.operators]**

```
template <typename Block>  
const_View<OutputType, unspecified>  
operator () (  
    const_View<InputType, Block> source)  
VSIP_THROW((std::bad_alloc));
```

Requires: ReturnMechanism must be `by_value`. InputType and OutputType must obey the appropriate row of [signal.fft]/4. `source.size()` and `this->input_size()` are element-conformant.

Returns: The fast Fourier transform of `source`. This is element-conformant to `this->output_size()` and has unit stride in dimension SD.

Throws: `std::bad_alloc` upon a memory allocation error.

Note: This function implements the functionality of the VSIPL functions `vsip_ccffftop_f`, `vsip_ccffftip_f`, `vsip_crffftop_f`, `vsip_rcffftop_f`, `vsip_ccffft2dop_f`, `vsip_ccffft2dip_f`, `vsip_crffft2dop_f`, `vsip_rcffft2dop_f`, `vsip_ccffft3dop_f`, `vsip_ccffft3dip_f`, `vsip_crffft3dop_f`, and `vsip_rcffft3dop_f`. Unlike VSIPL, there are no restrictions on strides.

```
template <typename Block0,
          typename Block1>
View<OutputType, Block1>
operator ( (
    const_View<InputType, Block0> source,
    View<OutputType, Block1>      destination)
VSIP_NOTHROW;
```

Requires: ReturnMechanism must be `by_reference`. InputType and OutputType must obey a row of [signal.fft]/4. `source.size()` and `this->input_size()` are element-conformant. `destination.size()` and `this->output_size()` are element-conformant. `source`'s block and `destination`'s block must not overlap.

Returns: The fast Fourier transform of `source`.

Effects: Stores the returned value in `destination`.

Note: This function implements the functionality of the VSIPL functions `vsip_ccffftop_f`, `vsip_ccffftip_f`, `vsip_crffftop_f`, `vsip_rcffftop_f`, `vsip_ccffft2dop_f`, `vsip_ccffft2dip_f`, `vsip_crffft2dop_f`, `vsip_rcffft2dop_f`, `vsip_ccffft3dop_f`, `vsip_ccffft3dip_f`, `vsip_crffft3dop_f`, and `vsip_rcffft3dop_f`. Unlike VSIPL, there are no restrictions on strides.

```
template <typename Block>
View<OutputType, Block>
operator ( (
```

11.3. [SIGNAL.FFTM] MULTIPLE FAST FOURIER TRANSFORMATIONS

```
View<OutputType, Block> source_and_destination)
VSIP_NOTHROW;
```

Requires: ReturnMechanism must be `by_reference`. InputType and OutputType must be the same. InputType and OutputType must obey a row of `[signal.fft]/4`. `source_and_destination.size()` and `this->input_size()` are element-conformant. `source_and_destination.size()` and `this->output_size()` are element-conformant. Block must be modifiable.

Returns: The fast Fourier transform of `source_and_destination`.

Effects: Stores the returned value in `source_and_destination`, which is overwritten.

Note: This function implements the functionality of the VSIPL functions `vsip_ccffftop_f`, `vsip_ccffftip_f`, `vsip_crffftop_f`, `vsip_rcffftop_f`, `vsip_ccfft2dop_f`, `vsip_ccfft2dip_f`, `vsip_crfft2dop_f`, `vsip_rcfft2dop_f`, `vsip_ccfft3dop_f`, `vsip_ccfft3dip_f`, `vsip_crfft3dop_f`, and `vsip_rcfft3dop_f`.

11.3 Multiple fast Fourier transformations

[**signal.fft**]

- 1 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. [*Note:* Stacked FFTs and vector FFTs are alternate names for multiple FFTs. Single fast Fourier transforms are specified in `[signal.fft]`.]
- 2 All VSIPL 1.1 API complexity requirements for multiple FFTs are incorporated by reference.
- 3 [*Note:* For mathematical descriptions of multiple FFTs, see the VSIPL description of `vsip_ccfftmop_f`, `vsip_crfftmop_f`, `vsip_rcfftmop_f`, and `vsip_ccfftmop_create_f`.]
- 4 For multiple transformations of subsets of the entire `Matrix`, `Fftm` supports different computations dependent on the input element type, output element type, a special dimension (“sd”), and a special direction. They must satisfy one of these criteria:

input type / output type	sd	direction	input size	output size
complex<T> / complex<T>	0 or 1	forward	$M \times N$	$M \times N$
	0 or 1	inverse	$M \times N$	$M \times N$
T / complex<T>	0	forward	$M \times N$	$M \times (N/2 + 1)$
	1	forward	$M \times N$	$(M/2 + 1) \times N$
complex<T> / T	0	inverse	$M \times (N/2 + 1)$	$M \times N$
	1	inverse	$(M/2 + 1) \times N$	$M \times N$

M and N indicate `length_types` indicating the number of rows and columns, respectively. [Note: To simplify the table, a row does not repeat a column entry if it is the same as in the previous row.]

T must be a type such that `complex<T>` can be instantiated. For a dimension that has different input and output sizes, the larger dimension size Q must be even so $Q/2 + 1$ is integral. The special dimension (“sd”) indicates in which dimension the multiple FFTs should occur. Dimension 0 (or `row`) indicates row-wise FFTs should occur. Dimension 1 (or `col`) indicates column-wise FFTs should occur.

[Example: An `Fftm` using `Matrixes` of `cscalar_fs` for both input and output requires the input and output `Matrixes` to have the same size $M \times N$. An `Fftm` using `Matrixes` with input type `scalar_f` and output type `cscalar_f` and special dimension one requires the input `Matrix` to have size $M \times N$ and the output to have size $(M/2 + 1) \times N$. M must be even.]

[Note: The special cases for non-complex input or output type permits reducing the computation time by about a factor of two.] For the specified dimension, the complex views for these cases conceptually have length N but only the first $\lfloor N/2 \rfloor + 1$ must be supplied or will be returned, as appropriate. The other values are specified by the identity $x_n = x_{N-n}^*$ for $\lfloor N/2 \rfloor < n < N$. For these complex views, the complex value at index position 0 represents the zero (DC) frequency and must have zero imaginary part. The complex value at index position $N/2$ represents the Nyquist frequency (one half the sample rate value) and must also be non-complex.

```
namespace vsip {
    int const fft_fwd = unspecified;
    int const fft_inv = unspecified;

    template <typename InputType, typename OutputType,
              int SD = row, int direction = fft_fwd,
              return_mechanism_type ReturnMechanism = by_value,
              unsigned number_of_times = 0, alg_hint_type ah = alg_time>
    class Fftm {
    public:
        // constructor, copies, assignments, and destructor
    
```

11.3.1. [SIGNAL.FFTM.CONSTANTS] CONSTANTS

```
Fftm(Domain<2> const&, scalar_f)
    VSIP_THROW((std::bad_alloc));
Fftm(Fftm const&) VSIP_NOTHROW;
Fftm& operator=(Fftm const&) VSIP_NOTHROW;
~Fftm() VSIP_NOTHROW;

// accessors
Domain<2> const& input_size() const
    VSIP_NOTHROW;
Domain<2> const& output_size() const
    VSIP_NOTHROW;
scalar_f scale() const VSIP_NOTHROW;
bool forward() const VSIP_NOTHROW;

// operators
// if ReturnMechanism is by_value:
template <typename Block>
const_Matrix<OutputType, unspecified>
operator() (const_Matrix<InputType, Block>)
    VSIP_THROW((std::bad_alloc));

// if ReturnMechanism is by_reference:
template <typename Block0, typename Block1>
Matrix<OutputType, Block1>
operator() (const_Matrix<InputType, Block0>,
           Matrix<OutputType, Block1>)
    VSIP_NOTHROW;
// if ReturnMechanism is by_reference and
// InputType == OutputType:
template <typename Block>
Matrix<OutputType, Block>
operator() (Matrix<OutputType, Block>)
    VSIP_NOTHROW;
};
}
```

- 5 The template `Fftm` class has seven template parameters and eight specializations corresponding to the above table. Specifications for the function and data members of these specializations are the same so they are presented only once below using `InputType` for the input type and `OutputType` for the output type.

11.3.1 Constants

[[signal.fft.constants](#)]

- 1 The constants `fft_fwd` and `fft_inv` are specified in [[signal.fft.constants](#)].

11.3.2 *Template parameters***[signal.fft.template]**

- 1 `InputType` and `OutputType` must obey the table above. The only specializations which must be supported have `T` equal to `scalar_f`. An implementation is permitted to prevent other instantiations. [*Note:* If `T` is `scalar_f`, `complex<T>` is `cscalar_f`.]

```
int SD
```

Requires: `0 <= SD && SD < 2`.

Note: If `SD == 0`, row-wise FFTs will occur. If `SD == 1`, column-wise FFTs will occur.

```
int direction
```

Requires: `direction == fft_fwd || direction == fft_inv`.

Note: If `InputType` and `OutputType` differ, this value is ignored. If `InputType` and `OutputType` are the same, this indicates the direction of the transforms, i.e., forward or inverse transforms.

- 2 [*Note:* The `return_mechanism_type` values indicate whether operators yield values by returning them by value while others use an “output” parameter. `by_value` and `by_reference` respectively describe each.]

```
unsigned number_of_times
```

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

```
alg_hint_type ah
```

Requires: A `alg_hint_type` value.

Note: This value indicates how an implementation should optimize its computation or resource use.

11.3.3 Constructors, copy, assignment, and destructor [signal.fftconstructors]

```
Fftm(
    Domain<2> const& dom,
    scalar_f         scale)
VSIP_THROW((std::bad_alloc));
```

Requires: dom must obey [signal.fft]/4 as determined by the template arguments.

Effects: Constructs an object of class `Fftm`.

Postconditions: `this->input_size()` and `this->output_size()` correspond to the appropriate row in [signal.fft]/4. `this->scale() == scale`.

Throws: `std::bad_alloc` upon a memory allocation error.

Note: This function implements part of the functionality of the VSIPL functions `vsip_ccfftmap_create_f`, `vsip_ccfftmip_create_f`, `vsip_crfftmop_create_f`, and `vsip_rcfftmop_create_f`.

11.3.4 Accessors

[signal.fftaccessors]

```
Domain<2> const&
input_size() const VSIP_NOTHROW;
```

Returns: A domain object with first indices of zero, unit strides, and size equal to the input size in the appropriate row in [signal.fft]/4.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
Domain<2> const&
output_size() const VSIP_NOTHROW;
```

Returns: A domain object with first indices of zero, unit strides, and size equal to the output size in the appropriate row in [signal.fft]/4.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
scalar_f scale() const VSIP_NOTHROW;
```

Returns: The scalar multiple, as specified in the object's constructor.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

```
bool
forward() const VSIP_NOTHROW;
```

Returns: An indication whether the transform is a forward or inverse transform.

Note: This function implements part of the behavior of the VSIPL function `vsip_fftn_attr_f`.

11.3.5 Operators

[`signal.fft.operators`]

```
template <typename Block>
const_Matrix<OutputType, unspecified>
operator() (
    const_Matrix<InputType, Block> source)
VSIP_THROW((std::bad_alloc));
```

Requires: ReturnMechanism must be `by_value`. InputType and OutputType must obey the appropriate row of [`signal.fft`]/4. `source.size()` and `this->input_size()` are element-conformant.

Returns: The multiple fast Fourier transforms of `source` which is element-conformant and using `this->output_size()` with unit stride in dimension `SD`. If `SD == row`, a separate transform applies to each row of `source`. If `SD == col`, a separate transform applies to each column of `source`.

Throws: `std::bad_alloc` upon a memory allocation error.

Note: This function implements the functionality of the VSIPL functions `vsip_ccfft mop_f`, `vsip_ccfft mip_f`, `vsip_crfft mop_f`, and `vsip_rcfft mop_f`. Unlike VSIPL, there are no restrictions on strides.

```
template <typename Block0,
          typename Block1>
Matrix<OutputType, Block1>
operator() (
    const_Matrix<InputType, Block0> source,
    Matrix<OutputType, Block1> destination)
VSIP_NOTHROW;
```

Requires: ReturnMechanism must be `by_reference`. InputType and OutputType must obey the appropriate row of `[signal.fft]/4`. `source.size()` and `this->input_size()` are element-conformant. `destination.size()` and `output_size()` are element-conformant. destination must be modifiable. source's block and destination's block must not overlap. Block1 must be modifiable.

Returns: The multiple fast Fourier transform of source in destination. If `SD == row`, a separate transform applies to each row of source. If `SD == col`, a separate transform applies to each column of source.

Effects: Stores the returned value in destination.

Note: This function implements the functionality of the VSIPL functions `vsip_ccfftmop_f`, `vsip_ccfftmip_f`, `vsip_crfftmop_f`, and `vsip_rcfftmop_f`. Unlike VSIPL, there are no restrictions on strides.

```
template <typename Block>
Matrix<OutputType, Block>
operator () (
    Matrix<OutputType, Block> source_and_destination)
VSIP_NOTHROW;
```

Requires: ReturnMechanism must be `by_reference`. InputType and OutputType must be the same. InputType and OutputType must obey a row of `[signal.fft]/4`. `source_and_destination.size()` and `this->input_size()` are element-conformant. `source_and_destination.size()` and `this->output_size()` are element-conformant. Block must be modifiable.

Returns: The multiple fast Fourier transform of `source_and_destination`. If `SD == row`, a separate transform applies to each row of `source_and_destination`. If `SD == col`, a separate transform applies to each column of `source_and_destination`.

Effects: Stores the returned value in `source_and_destination`, which is overwritten.

Note: This function implements the functionality of the VSIPL functions `vsip_ccfftmop_f`, `vsip_ccfftmip_f`, `vsip_crfftmop_f`, and `vsip_rcfftmop_f`.

1 A Convolution object performs decimated convolution filtering. [Note: For a mathematical description, see `vsip_conv1d_create_f` and `vsip_conv2d_create_f` in the VSIPL 1.1 API.]

```

namespace vsip {
    enum support_region_type { support_full, support_same,
                              support_min };
    enum symmetry_type { nonsym, sym_even_len_odd,
                       sym_even_len_even };

    template <template <typename, typename> class const_View,
              symmetry_type      symm,
              support_region_type supp,
              typename            T = VSIP_DEFAULT_VALUE_TYPE,
              unsigned            number_of_times = 0,
              alg_hint_type       ah = alg_time>
    class Convolution {
    public:
        // compile-time constants
        static symmetry_type const symmtry = symm;
        static support_region_type const supprt = supp;

        // constructors, copies, assignments, and destructors
        template <typename Block>
        Convolution(const_View<T, Block>,
                   Domain<const_View<T, Block>::dim> const&,
                   length_type decimation = 1)
            VSIP_THROW((std::bad_alloc));
        Convolution(Convolution const&) VSIP_NOTHROW;
        Convolution& operator=(Convolution const&)
            VSIP_NOTHROW;
        ~Convolution() VSIP_NOTHROW;

        // accessors
        Domain<const_View::dim> const& kernel_size() const
            VSIP_NOTHROW;
        Domain<const_View::dim> const& filter_order() const
            VSIP_NOTHROW;
        symmetry_type symmetry() const VSIP_NOTHROW;
        Domain<const_View::dim> const& input_size() const
            VSIP_NOTHROW;
        Domain<const_View::dim> const& output_size() const
            VSIP_NOTHROW;
        support_region_type support() const VSIP_NOTHROW;
        length_type decimation() const VSIP_NOTHROW;
    };
}

```

```
// Convolution
template <typename Block0, typename Block1>
typename ViewConversion<const_View, T, Block1>::view_type
operator() (const_View<T, Block0>,
           typename ViewConversion<const_View, T, Block1>::view_type)
    VSIP_NOTHROW;
};
}
```

2 `const_View::dim` indicates the dimensionality of `const_View`.

11.4.1 *Template parameters*

[**signal.convol.template**]

```
template <typename, typename> class const_View
```

Requires: `const_View<T, Block>` must be a valid C++ class for various values of `Block` including at least `Dense`. The class must support copy construction. The only specializations which must be supported are `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices of `const_View`.

```
typename T
```

Requires: The only specialization which must be supported has `T` the same as `scalar_f`. An implementation is permitted to prevent instantiation for other choices of `T`.

```
unsigned number_of_times
```

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

```
alg_hint_type ah
```

Note: This value indicates how an implementation should optimize its computation or resource use.

11.4.2 Enumerations

[signal.convolver.enum]

- 1 The enum `support_region_type` values indicate the *region of support* which is the number of output points for each dimension. `support_full` support has $\lfloor (N + M - 2)/D \rfloor + 1$ output points. `support_same` support has $\lfloor (N - 1)/D \rfloor + 1$ output points. `support_min` support has $\lfloor (N - 1)/D \rfloor - \lfloor (M - 1)/D \rfloor + 1$ output points. M specifies the length of one dimension of the kernel view, N specifies the length of one dimension of the input view, and D specifies the template decimation factor.
- 2 The enum `symmetry_type` values indicate symmetry and length for all dimensions of the view. `nonsym` indicates non-symmetric. `sym_even_len_odd` indicates even symmetric with odd length. `sym_even_len_even` indicates even symmetric with even length.

11.4.3 Constructors, copy, assignment, and destructor

[signal.convolver.constructors]

```
template <typename Block>
Convolution (
    const_View<T, Block>          filter_coeffs,
    Domain<const_View<T,Block>::dim> const input_size,
    length_type                   decimation)
VSIP_THROW((std::bad_alloc));
```

Requires: `decimation >= 1` . If `symmetry == nonsym`, `filter_coeffs.size() <= input_size.length()` . Otherwise, `filter_coeffs.size()*2 <= input_size.length()` .

Effects: Constructs an object of class `Convolution` .

Postconditions: `this->kernel_size()` returns a domain with first indices of zero, unit strides, and dimension lengths equal to the dimension sizes of `filter_coeffs`. `this->symmetry() == symmetry`. `this->input_size()` equals `input_size` but having first indices of zero and unit strides. `this->support() == support`.

Throws: `std::bad_alloc` upon memory allocation error.

11.4.4 Accessors

[signal.convolver.accessors]

```
Domain<const_View::dim> const&
```

kernel_size() const
VSIP_NOTHROW;

Returns: A domain having, for each dimension, the same `length_type` as `filter_coeffs`'s domain but having first indices of zero and unit strides.

Domain<const_View::dim> const&
filter_order() const
VSIP_NOTHROW;

Returns: `this->kernel_size()`.

`symmetry_type`
symmetry() const VSIP_NOTHROW;

Returns: `symmetry`.

Domain<const_View::dim> const&
input_size() const
VSIP_NOTHROW;

Returns: A domain with first indices of zero and unit strides indicating the required size of the operator's input view.

Domain<const_View::dim> const&
output_size() const
VSIP_NOTHROW;

Returns: A domain with first indices of zero and unit strides indicating the size of the operator's result.

`support_region_type`
support() const VSIP_NOTHROW;

Returns: `support`.

`length_type`
decimation() const VSIP_NOTHROW;

Returns: The output decimation factor.

11.4.5 Convolution operators

[signal.convol.operators]

```

template <typename Block0,
          typename Block1>
typename ViewConversion<View, T, Block1>::view_type
operator () (
    const_View<T, Block>                                v,
    typename ViewConversion<const_View, T, Block1>::view_type out)
VSIP_NOTHROW;

```

Requires: The domain of `v` and `this->input_size()` must be element-conformant. The domain of `out` and `this->output_size()` must be element-conformant. `out` must be modifiable. `v` and `out` must not overlap.

Returns: `out`.

Effects: The convolution of the constructor argument `filter_coeffs` and `v` as specified by the VSIP 1.1 API for `vsip_convolve1d_f` is stored in `out`.

11.5 Correlations

[signal.correl]

1 A Correlation object computes correlations between a reference view and a data view. [Note: For a mathematical description, see `vsip_corr1d_create_f` and `vsip_corr2d_create_f` in the VSIP 1.1 API.]

```

namespace vsip {

    enum matrix_type { biased, unbiased };

    template <template <typename, typename> class const_View,
              support_region_type supp,
              typename T = VSIP_DEFAULT_VALUE_TYPE,
              unsigned number_of_times = 0,
              alg_hint_type ah = alg_time>
    class Correlation {
    public:
        // compile-time constants and declarations
        static support_region_type const supprt = supp;

        // constructors, copies, assignments, and destructor
        Correlation(Domain<const_View::dim> const&,
                   Domain<const_View::dim> const&)

```

```
VSIP_THROW((std::bad_alloc));
Correlation(Correlation const&) VSIP_NOTHROW;
Correlation& operator=(Correlation const&)
    VSIP_NOTHROW;
~Correlation() VSIP_NOTHROW;

// accessors
Domain<const_View::dim> const& reference_size() const
    VSIP_NOTHROW;
Domain<const_View::dim> const& input_size() const
    VSIP_NOTHROW;
Domain<const_View::dim> const& output_size() const
    VSIP_NOTHROW;
support_region_type support() const VSIP_NOTHROW;

// Correlation
template <typename Block0, typename Block1,
          typename Block2>
typename ViewConversion<const_View, T, Block2>::view_type
operator()(matrix_type output_bias,
          const_View<T, Block0>, const_View<T, Block1>,
          typename ViewConversion<const_View, T, Block2>::view_type)
    VSIP_NOTHROW;
};
}
```

2 `const_View::dim` abbreviates the dimensionality of `const_View`.

11.5.1 Template parameters

[**signal.correl.template**]

```
template <typename, typename> class const_View
```

Requires: `const_View<T, Block>` must be a valid C++ class for various values of `Block` including at least `Dense`. The class must support copy construction. The only specializations which must be supported are `const_View` the same as `const_Vector` or `const_Matrix`. An implementation is permitted to prevent instantiation for other choices of `const_View`.

```
typename T = VSIP_DEFAULT_VALUE_TYPE
```

Requires: The only specialization which must be supported has `T` the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of `T`.

unsigned number_of_times

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

alg_hint_type ah

Note: This value indicates how an implementation should optimize its computation or resource use.

11.5.2 Constructors, copy, assignment, and destructor [signal.correl-constructors]

```
Correlation(
    Domain<const_View::dim> const& reference_size,
    Domain<const_View::dim> const& input_size)
VSIP_THROW((std::bad_alloc));
```

Requires: For each dimension *d*, `reference_size[d].length() <= input_size[d].length()`.

Effects: Constructs an object of class `Correlation`.

Postconditions: `this->reference_size()` equals `reference_size` but having first indices of zero and unit strides. `this->input_size() == input_size.this->support() == supprt`.

Throws: `std::bad_alloc` upon memory allocation error.

11.5.3 Accessors [signal.correl.accessors]

```
Domain<const_View::dim> const&
reference_size() const
VSIP_NOTHROW;
```

Returns: A domain having, for each dimension, the same `size()` as the `reference_size` domain given to the constructor but having first indices of zero and unit strides.

```
Domain<const_View::dim> const&
input_size() const VSIP_NOTHROW;
```

Returns: A domain with first indices of zero and unit strides indicating the required size of the operator's input view.

```
Domain<const_View::dim> const&
output_size() const
VSIP_NOTHROW;
```

Returns: A domain with first indices of zero and unit strides indicating the size of the operator's result.

```
support_region_type
support() const VSIP_NOTHROW;
```

Returns: `supprt`.

11.5.4 Correlation operators

[[signal.correl.operators](#)]

```
template <typename Block0,
          typename Block1,
          typename Block2>
typename ViewConversion<const_View, T, Block2>::view_type
operator () (
    bias_type                                     output_bias,
    const_View<T, Block0>                        reference_view,
    const_View<T, Block1>                        v,
    typename ViewConversion<const_View, T, Block2>::view_type out)
VSIP_NOTHROW;
```

Requires: The domain of `reference_view` and `this->reference_size()` must be element-conformant. The domain of `v` and `this->input_size()` must be element-conformant. The domain of `out` and `this->output_size()` are element-conformant. `out` must be modifiable. `out` cannot overlap `reference_view` or `v`.

Returns: `out`.

Effects: The correlation of the constructor argument `reference_view` and `v` as specified by the VSIP 1.1 API for `vsip_correlate1d.f` is stored in `out`. If `output_bias == biased`, then biased correlation estimates are stored. If the constructor `output_bias == unbiased`, then unbiased correlation estimates are stored.

1 These functions create `const_Vectors` of window weights.

```
namespace vsip {
    const_Vector<scalar_f, unspecified>
    blackman(length_type) VSIP_THROW((std::bad_alloc));

    const_Vector<scalar_f, unspecified>
    cheby(length_type, scalar_f)
        VSIP_THROW((std::bad_alloc));

    const_Vector<scalar_f, unspecified>
    hanning(length_type) VSIP_THROW((std::bad_alloc));

    const_Vector<scalar_f, unspecified>
    kaiser(length_type, scalar_f) VSIP_NOTHROW;
}
```

```
const_Vector<scalar_f, unspecified>
blackman(length_type len)
VSIP_THROW((std::bad_alloc));
```

Requires: `len > 1`.

Returns: A `const_Vector` initialized with Blackman window weights and having length `len`.

Throws: `std::bad_alloc` upon memory allocation error.

Note: The function corresponds to VSIPL function `vsip_vcreate_blackman_f`. See its description for the mathematical formula.

```
const_Vector<scalar_f, unspecified>
cheby(length_type len, scalar_f ripple)
VSIP_THROW((std::bad_alloc));
```

Requires: `len > 1`.

Returns: A `const_Vector` initialized with Dolph-Chebyshev window weights and having length `len`.

Throws: `std::bad_alloc` upon memory allocation error.

Note: The function corresponds to VSIPL function `vsip_vcreate_cheby_f`. See its description for the mathematical formula.

```
const_Vector<scalar_f, unspecified>  
hanning(length_type len) VSIP_THROW((std::bad_alloc));
```

Requires: `len > 1`.

Returns: A `const_Vector` initialized with Hanning window weights and having length `len`. The formula is the same as for `vsip_vcreate_hanning_f`.

Throws: `std::bad_alloc` upon memory allocation error.

Note: The function corresponds to VSIPL function `vsip_vcreate_hanning_f`. See its description for the mathematical formula.

```
const_Vector<scalar_f, unspecified>  
kaiser(length_type len, scalar_f beta) VSIP_NOTHROW;
```

Requires: `len > 1`.

Returns: A `const_Vector` initialized with Kaiser window weights with transition width parameter `beta` and having length `len`.

Note: The function corresponds to VSIPL function `vsip_vcreate_kaiser_f`. See its description for the mathematical formula.

11.7 FIR filters

[[signal.fir](#)]

- 1 The template class `Fir` implements finite impulse response filters. [*Note:* For a mathematical description, see `vsip_fir_create_f` in the VSIPL 1.1 API specifications.]
- 2 FIR filter objects support filtering long (semi-infinite) data streams by storing internal state information. This state information is incorporated from the VSIPL 1.1 API by reference.

```
namespace vsip {  
  
    enum obj_state { state_no_save, state_save };  
  
    template <typename T = VSIP_DEFAULT_VALUE_TYPE,  
              symmetry_type symme = nonsym,  
              obj_state contin_filtering = state_save,  
              unsigned number_of_times = 0,
```

```

    alg_hint_type ah = alg_time>
class Fir {
public:
    // compile-time constants
    static symmetry_type const symmetry = symme;
    static obj_state const continuous_filter
        = contin_filtering;

    // constructor, copies, assignments, and destructor
    template <typename Block>
    Fir(const_Vector<T, Block>, length_type, length_type = 1)
        VSIP_THROW((std::bad_alloc));
    Fir(Fir const&) VSIP_NOTHROW;
    Fir& operator=(Fir const&) VSIP_NOTHROW;
    ~Fir() VSIP_NOTHROW;

    // accessors
    length_type kernel_size() const VSIP_NOTHROW;
    length_type filter_order() const VSIP_NOTHROW;
    symmetry_type symmetry() const VSIP_NOTHROW;
    length_type input_size() const VSIP_NOTHROW;
    length_type output_size() const VSIP_NOTHROW;
    obj_state continuous_filtering() const VSIP_NOTHROW;
    length_type decimation() const VSIP_NOTHROW;

    // operators
    template <typename Block0, typename Block1>
    length_type operator()(const_Vector<T, Block0>,
                          Vector<T, Block1>)
        VSIP_NOTHROW;
    void reset() VSIP_NOTHROW;
};
}

```

11.7.1 Enumeration

[[signal.fir.enum](#)]

- 1 The enum `obj_state` value `state_save` indicates the filter will be used repeatedly on consecutive input segments, each having the same length, of a semi-infinite data stream. Thus, the filter needs to save input state information between operations on input. `state_no_save` indicates the filter will operate on independent input segments.

11.7.2 Template parameters

[[signal.fir.template](#)]

typename T

Requires: The only specializations which must be supported are T the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent instantiation for other choices of T .

`unsigned number_of_times`

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

`alg_hint_type ah`

Note: This value indicates how an implementation should optimize its computation or resource use.

11.7.3 Constructors, copy, assignment, and destructor [signal.fir.constructors]

```
template <typename Block>
Fir(
    const_Vector<T, Block> kernel,
    length_type          input_size,
    length_type          decimation = 1)
VSIP_THROW((std::bad_alloc));
```

Requires: Let M be the kernel order. $M \geq 1$. `kernel.size() = M + 1`
 if `symm == nonsym` . `kernel.size() = [(M + 1)/2]`
 if `symmetry == sym_even_len_odd` || `symmetry == sym_even_len_even`. $M \geq \text{decimation}$. `input_size $\geq M$` .
`decimation ≥ 1` .

Effects: Constructs an object of class `Fir`.

Postconditions: If `continuous_filtering == state_save`, then the save state will be stored in the object and initialized to zeros. `this->kernel_size() == M`.
`this->symmetry() == symmetry` . `this->input_size() == input_size` .
`this->continuous_filtering() == continuous_filtering` . `this->decimation() == decimation`.

Throws: `std::bad_alloc` upon memory allocation error.

Note: The object must store any values from `kernel` separately from the `kernel` argument. This function implements part of the functionality of the VSIPL functions `vsip_fir_create_f` and `vsip_cfir_create_f`.

11.7.4 Accessors

[signal.fir.accessors]

```
length_type
kernel_size() const VSIP_NOTHROW;
```

Returns: $M + 1$.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
length_type
filter_order() const VSIP_NOTHROW;
```

Returns: `kernel_size()`.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
symmetry_type
symmetry() const VSIP_NOTHROW;
```

Returns: The `symmetry_type` template argument.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
length_type
input_size() const VSIP_NOTHROW;
```

Returns: The required size of the operator's input vector.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
length_type
output_size() const VSIP_NOTHROW;
```

Returns: The size of the filtering operator's result vector, i.e., `ceil(input_size()/decimation)`.

Note: The returned value may exceed the number of computed values in the result vector by one. This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
obj_state
```

```
continuous_filtering() const VSIP_NOTHROW;
```

Returns: `state_save` iff the save state is stored in the object.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

```
length_type
```

```
decimation() const VSIP_NOTHROW;
```

Returns: The output decimation factor.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_getattr_f` and `vsip_cfir_getattr_f`.

11.7.5 Filtering and state reset operators

[**signal.fir.operators**]

```
template <typename Block0,
          typename Block1>
```

```
length_type
```

```
operator() (
```

```
    const_Vector<T, Block> data,
    Vector<T, Block1>      out)
```

```
VSIP_NOTHROW;
```

Requires: The domain of `data` and `this->input_size()` must be element-conformant. `out` and `this->output_size()` must be element-conformant. `out` must be modifiable. `data` and `out` must not overlap.

Returns: The number of computed values.

Effects: The result of applying the FIR filter to the `data` is stored in `out`.

Postconditions: If `this->continuous_filtering() == state_save`, the save state is updated. The returned vector has `this->size() == output_size()`.

Note: This function implements part of the functionality of the VSIPL functions `vsip_firflt_f` and `vsip_cfirflt_f`.

```
void
```

```
reset() VSIP_NOTHROW;
```

Postconditions: If `this->continuous_filtering() == state_save`, then the save state is initialized to zeros.

Note: This function implements part of the functionality of the VSIPL functions `vsip_fir_reset_f` and `vsip_cfir_reset_f`.

11.8 IIR filters

[signal.iir]

- 1 The template class `Iir` implements infinite impulse response filters. [*Note:* For a mathematical description, see `vsip_iir_create_f` in the VSIPL 1.1 API.]
- 2 IIR filter objects support filtering long (semi-infinite) data streams by storing internal state information. This state information is incorporated from the VSIPL 1.1 API by reference.

```

namespace vsip {
  template <typename T = VSIP_DEFAULT_VALUE_TYPE,
           obj_state contin_filtering = state_save,
           unsigned number_of_times = 0, alg_hint_type ah = alg_time>
  class Iir {
  public:
    // compile-time constants
    static obj_state const continuous_filtering
      = contin_filtering;

    // constructor, copies, assignments, and destructor
    template <typename Block0, typename Block1>
    Iir(const_Matrix<T, Block0>,
        const_Matrix<T, Block1>,
        length_type const)
      VSIP_THROW((std::bad_alloc));
    Iir(Iir const&) VSIP_THROW((std::bad_alloc));
    Iir& operator=(Iir const&) VSIP_THROW((std::bad_alloc));
    ~Iir() VSIP_NOTHROW;

    // accessors
    length_type kernel_size() const VSIP_NOTHROW;
    length_type filter_order() const VSIP_NOTHROW;
    length_type input_size() const VSIP_NOTHROW;
    length_type output_size() const VSIP_NOTHROW;
    obj_state continuous_filtering() const VSIP_NOTHROW;

    // operators
    template <typename Block0, typename Block1>
    Vector<T, Block1> operator()(const_Vector<T, Block0>,
                                Vector<T, Block1>)
      VSIP_NOTHROW;
    void reset() VSIP_NOTHROW;
  };
}

```

- 3 [*Note:* See [signal.fir.enum] for a description of enum `obj_state`.]

*11.8.1 Template parameters***[signal.iir.template]**

typename T

Requires: The only specialization which must be supported is T the same as `scalar_f`. An implementation is permitted to prevent instantiation for other choices of T.

unsigned number_of_times

Note: This value indicates the anticipated number of times the object will be used. A value of zero indicates semi-infinity, i.e., many times.

alg_hint_type ah

Note: This value indicates how an implementation should optimize its computation or resource use.

11.8.2 Constructors, copy, assignment, and destructor **[signal.iir.constructors]**

```
template <typename Block0,
          typename Block1>
Iir(
    const_Matrix<T, Block0> B,
    const_Matrix<T, Block1> A,
    const length_type      input_size)
VSIP_THROW((std::bad_alloc));
```

Requires: For an order $2M$ filter, `B.size(0) = M`, `B.size(1) = 3`, `A.size(0) = M`, and `A.size(1) = 2`. `input_size ≥ 2M`.

Effects: Constructs an object of class `Iir`.

Postconditions: If `continuous_filtering == state_save`, then the save state will be stored in the object and initialized to zeros. `this->kernel_size() = 2M`. `this->input_size() == input_size`. `this->continuous_filtering() == continuous_filtering`.

Note: The object must store any values from B and A separately from the arguments. This function implements part of the functionality of the VSIPL function `vsip_iir_create_f`.

11.8.3 Accessors

[signal.iir.accessors]

```
length_type
kernel_size() const VSIP_NOTHROW;
```

Returns: $2M$.**Note:** This function implements part of the functionality of the VSIPL function `vsip_iir_getattr.f`.

```
length_type
filter_order() const VSIP_NOTHROW;
```

Returns: `this->kernel_size()`.**Note:** This function implements part of the functionality of the VSIPL function `vsip_iir_getattr.f`.

```
length_type
input_size() const VSIP_NOTHROW;
```

Returns: The required size of the operator's input vector.**Note:** This function implements part of the functionality of the VSIPL function `vsip_iir_getattr.f`.

```
length_type
output_size() const VSIP_NOTHROW;
```

Returns: `this->input_size()`.**Note:** This function implements part of the functionality of the VSIPL function `vsip_iir_getattr.f`.

```
obj_state
continuous_filtering() const VSIP_NOTHROW;
```

Returns: `state_save` iff the save state is stored in the object.**Note:** This function implements part of the functionality of the VSIPL function `vsip_iir_getattr.f`.

11.8.4 *Filtering and state reset operators***[signal.iir.operators]**

```

template <typename Block0,
          typename Block1>
Vector<T, Block1>
operator () (
    const_Vector<T, Block0> data,
    Vector<T, Block1> out)
VSIP_NOTHROW;

```

Requires: The domain of `data` and `this->input_size()` must be element-conformant. `out` and `this->output_size()` must be element-conformant. `data` and `out` must not overlap.

Returns: `out`.

Effects: The result of applying the IIR filter to the data is stored in `out`.

Postconditions: If `continuous_filtering() == state_save`, the save state is updated. The returned vector has `size() == this->output_size()`.

Note: This function implements part of the functionality of the VSIPL function `vsip_iirflt_f`.

```

void
reset () VSIP_NOTHROW;

```

Postconditions: If `this->continuous_filtering() == state_save`, then the save state is initialized to zeros.

Note: This function implements part of the functionality of the VSIPL function `vsip_iir_reset_f`.

11.9 *Histograms***[signal.histo]**

```

namespace vsip {
    template <template <typename, typename> class const_View = const_Vector,
             typename T = VSIP_DEFAULT_VALUE_TYPE>
    class Histogram {
    public:
        // constructor and destructor
        Histogram(T, T, length_type) VSIP_THROW((std::bad_alloc));
        ~Histogram() VSIP_NOTHROW;

        // operator()

```

```

template <typename Block>
const_Vector<scalar_i, unspecified> operator()
    (const_View<T, Block>, bool = false) VSIP_NOTHROW;
};
}

```

- 1 The template class `Histogram` supports computing the histogram of its input data.
- 2 The only specializations of `Histogram` which must be supported are for `const_View` the same as `const_Vector` or `const_Matrix` and `T` the same as `scalar_f` or `scalar_i`. An implementation is permitted to prevent other instantiations.

11.9.1 Constructor and destructor

[[signal.histo.constructors](#)]

```

Histogram(
    T          min_value,
    T          max_value,
    length_type num_bin)
VSIP_THROW((std::bad_alloc));

```

Requires: `min_value < max_value.num_bin >= 3`.

Effects: Constructs an object of class `Histogram` with zero values in all `num_bin` bins.

Throws: `std::bad_alloc` if the memory allocation for the returned `const_Vector` fails.

11.9.2 Histogram operators

[[signal.histo.operators](#)]

```

template <typename Block>
const_Vector<scalar_i, unspecified>
operator () (
    const_View<T, Block> data,
    bool                accumulate = false)
VSIP_NOTHROW;

```

Returns: A histogram of data with `num_bin-2` bins distributed linearly over the range `[min_value,max_value)`. Bins with indices 0 and `num_bin-1` accumulate values less than `min_value` and greater than or equal to `max_value`, respectively. If `accumulate` is true, values from

data increment previously computed values. If `accumulate` is `false`, previously computed values are zeroed before values from data increment these values.

Effects: Stores the returned values internally. `num_bin`, i.e., the output length specified in the constructor.

Note: This functionality is similar to VSIPL functions `vsip_vhisto_f`, `vsip_vhisto_i`, `vsip_mhisto_f`, and `vsip_mhisto_i`. See the VSIPL 1.1 API for a mathematical description.

11.10 Frequency swap functions

[`signal.freqswap`]

```
namespace vsip {
    template <template <typename, typename> class const_View,
              typename T, typename Block>
    const_View<T, unspecified>
    freqswap(const_View<T, Block>) VSIP_NOTHROW;
}
```

- 1 The only specializations of `freqswap` which must be supported are for `const_View` the same as `const_Vector` or `const_Matrix` and `T` the same as `scalar_f` or `cscalar_f`. An implementation is permitted to prevent other instantiations.

```
template <template <typename, typename> class const_View,
          typename T,
          typename Block>
const_View<T, unspecified>
freqswap(const_View<T, Block> source) VSIP_NOTHROW;
```

Requires: The block of the returned `const_View` must either not overlap `source.block()` or be exactly the same.

Returns: If `const_View` is `const_Vector`, a `const_Vector` with the two halves of `source` swapped. If `const_View` is `const_Matrix`, a `const_Matrix` with the upper left and lower right quadrants of `source` swapped. Given an odd-length, the left half has one more value than the right-half.

Note: This functionality is similar to VSIPL functions `vsip_vfreqswap_f`, `vsip_cvfreqswap_i`, `vsip_mfreqswap_f`, and `vsip_cmfreqswap_i`. See the VSIPL 1.1 API for a mathematical description.

This document reflects version 1.02 of the VSIPL++ specification approved by the VSIPL Forum on 3rd June 2010. This document has been reviewed by the Forum.

Version	Date Approved	Changes
1.0	9 Mar 2005	Initial VSIPL++ specification.
1.01	23 Jun 2005	Change all-capital class, enumeration, and variable names to avoid potential macro collisions with C libraries.
1.02	3 Jun 2010	Additional overloads for view composite assignments (20). Correct sumsqval signature (45). In [domain.domainone.constructors], [view.dense.constructors], and [view.dense.userdata]: Indicate default arguments in function signatures. In [selgen.selection.scatter], [signal.convolver.constructors], [signal.convolver.operators], [signal.correl.operators], [signal.fir.operators], and [signal.iir.operators]. Remove reference to non-existent View::domain() and Block::domain() members. In [view.view] view requirements table, correct block type expression. In [support] and [support.types.domain] add enumeration whole`domain` type. In [view.tensor], [view.tensor.subview`types], and [view.tensor.subviews], change vector and matrix subview types to support compile-time fixed dimensions and whole-domains. In [view.tensor], [view.tensor.subview`types], and [view.tensor.transpose], change transpose subview types to support compile-time permutations of dimensions.