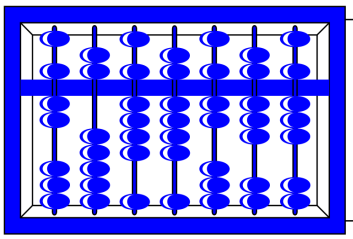


NASoftware Limited

Incorporating InfoSAR

VSIPL
Reference Manual

VSIPL/Ref [3.20.14]

Release 3.20.14
February 2021

This document is derived from the VSIPL API 1.01 Specification document written by David A. Schwartz (HRL Laboratories, LLC), Randall R. Judd (Space and Naval Warfare Systems Center, San Diego), William J. Harrod (Silicon Graphics Inc./Cray Research) and Dwight P. Manley (Compaq Computer Corp./Digital Equipment Corp.), approved by the VSIPL Forum (27 March 2001), and available from `http://www.vsipl.org/`.

This material is based in part upon work supported by Ft. Huachuca/DARPA under contract No. DABT63-96-C-0060. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Ft. Huachuca/DARPA.

Contents

Chapter 1. VSIPL Introduction

1.1 Introduction to VSIPL

The purpose of the Vector, Signal, and Image Processing Library (VSIPL) is to support portable, high performance application programs. The VSIPL specification is based upon existing libraries that have evolved and matured over decades of scientific and engineering computing. A layer of abstraction is added to support portability across diverse memory and processor architectures. The primary design focus of the specification has been embedded signal processing platforms. Enhanced portability of workstation applications is a side benefit.

1.1.1 Platform Requirements

VSIPL was designed so that it could be implemented on a wide variety of hardware. In order to use VSIPL functions on a given platform, a VSIPL compliant library must be available for the particular hardware and a tool-set (ANSI C compiler and linker) available for the operating system.

1.1.2 VSIPL Functionality

The VSIPL specification provides a number of functions to the programmer that support high performance numerical computation on dense rectangular arrays. These are organized in the VSIPL documentation according to category. The available categories include:

- Support
  - Library initialization and finalization
  - Object creation and interaction
  - Memory management
- Basic Scalar Operations
- Basic Vector Operations
- Random Number Generation
- Signal Processing
  - FFT operations
  - Filtering
  - Correlation and convolution
- Linear Algebra

- Basic matrix operations
- Linear system solution
- Least-squares problem solution

Although there are many functions in the VSIPL specification, not all functions are available in all libraries. The contents of a specific VSIPL library subset are defined in a profile. As of the completion of VSIPL 1.0 two profiles have been approved by the VSIPL Forum, referred to as the 'Core' and 'Core Lite' profiles. The 'Core' profile includes most of the signal processing and matrix algebra functionality of the library. The 'Core Lite' profile includes a smaller subset, suitable for vector-based signal processing applications. The VSIPL specification defines more functions than are present in either of these profiles.

This library implements the 'Core' profile with some extensions.

1.1.3 VSIPL Objects

The main difference between the VSIPL standard and existing libraries is a cleaner encapsulation of memory management through an 'object-based' design. In VSIPL, a block can be thought of as a contiguous area of memory for storage of data. A block consists of a data array, which is the memory used for data storage, and a block object, which is an abstract data type which stores information necessary for VSIPL to access the data array. VSIPL allows the user to construct a view of the data in a block as a vector, matrix, or higher dimensional object. A view consists of a block, which contains the data of interest, and a view object, which is an abstract data type that stores information necessary for VSIPL to access the data of interest.

Blocks and views are opaque: they can only be created, accessed and destroyed via library functions. Object data members are private to hide the details of non-portable memory hierarchy management. VSIPL library developers may hide information peculiar to their implementations in the objects in order to prevent the application programmer from accidentally writing code that is neither portable nor compatible.

Data arrays in VSIPL exist in one of two logical data spaces. These are the user data space, and VSIPL data space. VSIPL functions may only operate on data in VSIPL space. User supplied functions may only operate on data in user space. Data may be moved between these logical spaces. Depending on the specific implementation, this move may incur actual data movement penalties or may simply be a bookkeeping procedure. The user should consider the data in VSIPL space to be inaccessible except through VSIPL functions.

1.1.4 Other Features

Two versions of the library are described, referred to as development and performance libraries. These libraries operate to produce identical results with the exception of error reporting and timing. The performance version of the VSIPL library does not

provide any error detection or handling except in the case of memory allocation. Other programming errors under a VSIPL performance library may have unpredictable results, up to and including complete system crashes. The development library runs slower than the performance library but includes more error detection capabilities.

1.2 Basic VSIPL Concepts

1.2.1 General Library Design Principles

VSIPL supports high performance numerical computation on dense rectangular arrays, and incorporates the following well-established characteristics of existing scientific and engineering libraries:

1. Elements are stored in one-dimensional data arrays, which appear to the application programmer as a single contiguous block of memory.
2. Data arrays can be viewed as either real or complex, vectors or matrices.
3. All operations on data arrays are performed indirectly through view objects, each of which specifies a particular view of a data array with particular offset, length(s) and stride(s).
4. In general, the application programmer cannot combine operators in a single statement to evaluate expressions. Operators which return a scalar may be combined, but most operators will return a view type or are void and may not be combined.

Operators are restricted to views of a data array that can be specified by an offset, lengths and strides. Views that are more arbitrary are converted into these simple views by functions like gather and back again by functions like scatter. VSIPL does not support triangular or sparse matrices very well, though future extensions might address these. The main difference between the VSIPL and existing libraries is a cleaner encapsulation of the above principles through an 'object-based' design. All of the view attributes are encapsulated in opaque objects: such an object can only be created, accessed and destroyed via library functions, which references it via a pointer.

1.2.2 Memory Management

The management of memory is important to efficient algorithm development. This is especially true in embedded systems, many of which are memory limited. In VSIPL memory management is handled by the implementation. This section describes VSIPL memory management and how the user interacts with VSIPL objects.

Terminology

The terms *user data*, *VSIPL data*, *admitted*, and *released* are used throughout this document when describing memory allocation. It is important that the reader understands the terms that are defined in this section and in the Glossary.

Object Memory Allocation

All objects in VSIPL consist of abstract data types (ADT) that contain attributes defining the underlying data accessed by the object. Certain of the attributes are accessible to the application programmer via access functions; however, there may be any number of attributes assigned by the VSIPL library developer for internal use. Each time an object is defined, memory must be allocated for the ADT. All VSIPL objects are allocated by VSIPL library functions. There is no method by which the application programmer may allocate space for these objects outside of VSIPL. Most VSIPL objects are relatively small and of fixed size; however, some of the objects created for signal processing or linear algebra may allocate large workspaces.

Data Memory Allocation

A data array is an area of memory where data is stored. Data arrays in VSIPL exist in one of two logical data spaces. These are the user data space, and VSIPL data space. VSIPL functions may only operate on data in VSIPL space. User supplied functions may only operate on data in user space. Data may be moved between these logical spaces. Depending on the specific implementation, this move may incur actual data movement penalties or may simply be a bookkeeping procedure. The user should consider the data in VSIPL space to be inaccessible except through VSIPL functions.

A data array allocated by the application, using any method not part of the VSIPL standard, is considered to be a user data array. The application has a pointer to the user data array and knowledge of its type and size. Therefore the application can access a user data array directly using pointers, although it is not always correct to do so.

A data array allocated by a VSIPL function call is referred to as a VSIPL data array. The user has no proper method to retrieve a pointer to such a data array; it may only be accessed via VSIPL function calls.

Users may access data arrays in VSIPL space using an entity referred to as a block. The data array associated with a block is a contiguous series of elements of a given type. There is one block type for each type of data processed by VSIPL.

There are two categories of block: user blocks and VSIPL blocks. A user block is one that has been associated with a user data array. A VSIPL block is one that has been associated with a VSIPL data array. The data array referenced by the block is referred to as being 'bound' to the block. The user must provide a pointer to the associated data for a user block. The VSIPL library will allocate space for the data associated with a VSIPL block. Blocks can also be created without any data and then associated with data in user space. The process of associating user space data with a block is called *binding*. A block which does not have data bound to it may not be used, as there is no data to operate on.

A block that has been associated with data may exist in one of two states, admitted and released. The data in an *admitted* block is in the logical VSIPL data space, and the data in a *released* block is in the logical user data space. The process of moving data

from the logical VSIPL data space to the logical user data space is called admission; the reverse process is called release.

Data in an admitted block is owned by the VSIPL library, and VSIPL functions operate on this data under the assumption that the data will only be modified using VSIPL functions. VSIPL blocks are always in the admitted state. User blocks may be in an admitted state. User data in an admitted block shall not be operated on except by VSIPL functions. Direct manipulation of user data bound to an admitted block via pointers to the allocated memory is incorrect and may cause erroneous behaviour.

Data in a released block may be accessed by the user, but VSIPL functions should not perform computation on it. User blocks are created in the released state. The block must be admitted to VSIPL before VSIPL functions can operate on the data bound to the block. A user block may be admitted for use by VSIPL and released when direct access to the data is needed by the application program. A VSIPL block may not be released.

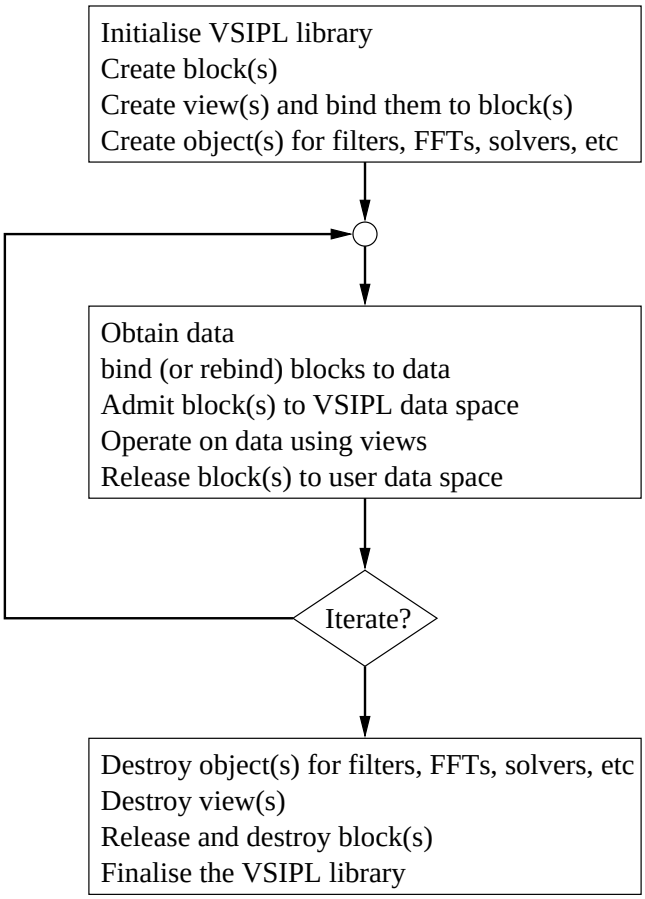
Blocks represent logically contiguous data areas in memory (physical layout is undefined for VSIPL space), but users often wish to operate on non-contiguous subsets of these data areas. To provide support for such operations, VSIPL requires that users operate on the data in a block through another object called a view. Views allow the user to specify non-contiguous subsets of a data array and inform VSIPL how the data will be accessed (for example, as a vector or matrix). When creating a vector view, the user specifies an offset into the block, a view length, and a stride value which specifies the number of elements (defined in the type of the block) to advance between each access. Thus, for a block whose corresponding data array contains four elements, a view with an offset value of zero, a stride of two, and a length of two represents a logical data set consisting of members zero and two of the original block. For a matrix view, stride and length parameters are specified in each dimension, and a single offset is specified. By varying the stride, row-major or column-major matrices can be created.

A block may have any number of views created on it; this allows the user to use vector views to access particular rows or columns of a matrix view, for example. Since the blocks are typed, views are also typed. However, because views also include usage information (e.g. vector or matrix), there are multiple view types for each block type corresponding to how the data will be accessed. These types are immutable; thus for example, a block cannot have both integer and float views associated with it. This would not be useful in any event because the data layout inside VSIPL space is vendor specific.

New views of a block may be created directly using a block object, or indirectly using a previously created view of the block. Except for finding the real or imaginary view of a complex view, all views may be created directly using the block object.

1.2.3 Structure of a VSIPL application

Although there are a number of ways to program an application, the basic VSIPL program consists of the following sequence:



A VSIPL program must initialize the VSIPL library with a call to `vsip_init` before calling any other VSIPL function. Any program that uses VSIPL and that terminates must call `vsip_finalize` before terminating. See the Support chapter for additional conditions and restrictions on these functions.

1.2.4 VSIPL Naming Conventions

While there is nothing to prevent a programmer from writing VSIPL-compatible functions, only those functions that are approved and included in formal VSIPL documentation are a part of VSIPL. Functions outside the standard should not use the VSIPL naming conventions in order to avoid confusion and application porting problems. In particular, names outside of VSIPL should not start with `vsip` or `vsipl`, in either upper or lower case.

Names for VSIPL types, objects and functions have the following format:

`vsip_`⟨*depth*⟩⟨*shape*⟩`basename_`⟨*precision–type*⟩

The shape qualifier specifies scalar, vector or matrix; the depth qualifier specifies real or complex; the precision–type qualifier specifies the data type (boolean, integer, or float) and its precision.

The depth qualifier is `r` or `c` for real or complex, respectively. The real qualifier is understood (not included as part of the name) if there can be no confusion. For a generic name `D` is used to indicate either real or complex.

The shape qualifier is `s`, `v` or `m` for scalar, vector or matrix, respectively. The scalar qualifier is understood (not included as part of the name) if there can be no confusion. For a generic name `S` is used to indicate any shape.

The precision–type qualifier is one of the following:

`i`	signed integer	`bl`	boolean
`si`	short signed integer	`vi`	vector index
`u`	unsigned integer	`mi`	matrix index
`f`	float		

This qualifier has no default value; it is only omitted on void functions. For a generic name `P` is used to indicate any precision–type.

For example, the generic function `vsip_DSmag_P` takes the magnitude (absolute value) of its argument. Specific instances of this function could include:

`vsip_mag_i`	(real) (scalar) integer
`vsip_cmag_f`	complex (scalar) float
`vsip_vmag_f`	vector of (real) floats
`vsip_cvmag_f`	vector of complex floats
`vsip_mmag_f`	matrix of (real) floats
`vsip_cmmag_f`	matrix of complex floats

For functions with arguments of different depths, shapes or types, the qualifiers may be repeated. For example the copy functions have two precision–type qualifiers corresponding to the data types of the source and destination arrays.

1.2.5 Non-standard Scalar Data Types

In general, VSIPL scalar data types correspond to particular C data types depending on the underlying implementation. However, ANSI C does not define boolean or complex scalar types, both of which are defined in VSIPL. This section summarises requirements for these data types.

Boolean Data Types

The VSIPL boolean data type (`bl`) is either true or false when used by a VSIPL function which sets or uses the boolean type. If a numeric vector or matrix is copied to a boolean vector or matrix then the value zero is copied to the boolean as false. Any other value

is copied as true. If a boolean vector or matrix is copied to a numeric vector or matrix then the value false is copied as a zero, and the value true is copied as positive one. If a VSIPL function returns a boolean scalar then a false is returned as zero and a true is non-zero. If a scalar is tested as boolean using a VSIPL function then a zero is tested as false and a non-zero is tested as true.

Complex Data Types

The definition of the complex scalar is available in public header files, and has the usual structure for complex data as normally defined in ANSI C programs. In general, users are encouraged to not use the structure directly, but to instead use VSIPL scalar functions for manipulating complex scalars. This should enhance portability of user code.

1.2.6 Data Array Layout

A user data array that is bound to a block has a particular required layout, depending on the type of the block. This section describes the required layout of the user data array for various block types. The application programmer must use the data array formats for user data. These formats allow portable input of user data into VSIPL and portable output of VSIPL results to the application.

For basic VSIPL types, the user data array is simply contiguous memory of the corresponding VSIPL type. This applies to floating-point, integer, boolean and vector index types.

For matrix index data, the user data array is contiguous memory of type `vsip_scalar_vi`; each matrix index element is two consecutive elements of type vsip_scalar_vi; the first element is the row, the second is the column. Note that the matrix index element in a user data array is not the same as `vsip_scalar_mi`.

For complex float or complex integer data, the user data array is either interleaved or split as described below. Both the interleaved and split formats are supported for user data. Note that the data format for complex user data arrays is not of type `vsip_cscalar_P`.

Interleaved The user data array is contiguous memory of type `vsip_scalar_P`. The complex element is two consecutive elements of type `vsip_scalar_P`. The first element is the real component and the second is the imaginary component.

Split The user data array consists of two contiguous memory regions of equal length, each of type `vsip_scalar_P`. The real and the imaginary regions are determined when the memory is bound to the block. A complex element consists of corresponding elements from the real and imaginary regions.

1.2.7 Errors and Restrictions

Many functions require that their arguments be **conformant**. This means that the objects passed have compatible attributes: for example, size and shape of matrices,

lengths of vectors or filter kernels.

If an argument is required to be **valid**, it means:

- a pointer is not `NULL`
- a flag is a member of the required enumerated type
- an object has been initialised and not destroyed.

Errors can occur for the following reasons:

1. an argument is outside the domain for calculation
2. over/underflow during calculation
3. failure to allocate memory
4. algorithm failure because of inappropriate data (as when a matrix does not have full rank)
5. arguments are invalid, out of range, or non-conformant.

Only errors of type 5 are regarded as fatal: in this case, the development version of the library will write a message to `stderr` and call `exit`.

Errors of types 3 and 4 are signalled through the return value of the function. A create function will return `NULL` if the allocation fails; functions with integer return codes use zero to indicate success.

The calling program is not alerted to errors of types 1 and 2.

1.3 Implementation-specific Details

1.3.1 Types

The following VSIPL base types are available:

```
vsip_scalar_i
vsip_scalar_si
vsip_scalar_u
vsip_scalar_bl    vsip_bool
vsip_scalar_vi    vsip_index
vsip_scalar_mi
vsip_scalar_f
vsip_cscalar_i
vsip_cscalar_si
vsip_cscalar_f
vsip_offset
vsip_stride
vsip_length
```

VSIPL also passes information around in abstract data types. These objects are opaque structures (implemented as incomplete typedefs) and they can only be created, accessed, and destroyed with library functions that reference them via a pointer. Some are used to describe the data layout in memory; others store information on filters, matrix decompositions, and so on. Some objects have a 'get attribute' function that allows the user access to the internal values.

The following structures for passing data are available:

```
vsip_block_f     vsip_vview_f     vsip_mview_f
vsip_block_i     vsip_vview_i     vsip_mview_i
vsip_block_si    vsip_vview_si    vsip_mview_si
vsip_block_bl    vsip_vview_bl
vsip_block_vi    vsip_vview_vi
vsip_block_mi    vsip_vview_mi
vsip_cblock_f    vsip_cvview_f    vsip_cmview_f
vsip_cblock_i    vsip_cvview_i    vsip_cmview_i
vsip_cblock_si   vsip_cvview_si   vsip_cmview_si
```

1.3.2 Symbols and Flags

The following symbolic constants are defined.

```
VSIP_MIN_SCALAR_F    VSIP_MAX_SCALAR_F
VSIP_MIN_SCALAR_I    VSIP_MAX_SCALAR_I
VSIP_MIN_SCALAR_SI   VSIP_MAX_SCALAR_SI
VSIP_MIN_SCALAR_BL   VSIP_MAX_SCALAR_BL
VSIP_MIN_SCALAR_VI   VSIP_MAX_SCALAR_VI
VSIP_TRUE
VSIP_FALSE
VSIP_PI
```

Other symbols are defined in enumerated types. The valid choices are listed with each function description.

1.3.3 Complex Variables

The preferred storage arrangement for complex data is split.

1.3.4 Hints

VSIPL provides the following mechanisms for the programmer to indicate preferences for optimisation: **they are all ignored** in the current implementation.

- Flags of the enumerated type `vsip_memory_hint` specified when allocating or creating some objects.

- Flags of the enumerated type `vsip_alg_hint` used to indicate whether algorithmic optimisation should minimise execution time, memory use, or maximise numerical accuracy.
- An indication of how many times an object will be used (filters and FFT's have such a parameter).

1.3.5 Notation

The following standard mathematical notation is used in the function descriptions.

$:=$	assignment operator
i	square root of -1
$\lvert x\rvert$	absolute value of the real number x
$\lvert z\rvert$	modulus of the complex number z
$\lfloor x\rfloor$	floor of the real number x (largest integer less than or equal to x)
$\lceil x\rceil$	ceiling of the real number x (smallest integer greater than or equal to x)
z^*	conjugate of the complex number z
M^{T}	transpose of the matrix M
M^{H}	Hermitian (conjugate transpose) of the complex matrix M

Note that in expressions i is always the square root of -1; vectors and matrices are indexed with j and k.

An elementwise operation on vectors will be written $C[j] := A[j] + B[j]$. Sometimes, the range of the index variable is not given explicitly; in such cases it is clear from the context that it runs over all the elements in the vectors and that the lengths of the vectors must be equal.

An M by N matrix has M rows and N columns.

Chapter 2. Getting the Best Performance

This section is a short guide for programmers using the NAS VSIPL Library. It contains explanations of library behavior, and tips on selecting the right storage options for your data to increase performance.

2.1 Version Information

Information about the version of the NAS VSIPL library you are using can be found in the comments at the top of the include file `vsip.h`. There is no way that a program can determine the library version at run-time.

2.2 Memory Alignment

The efficiency of many operations is improved if data within memory is correctly aligned on certain word boundaries. This is only of concern for memory allocated outside VSIPL and then bound to a VSIPL block — any storage allocated by the library via a create function is optimally aligned automatically.

Vectors and matrices can be loaded and stored faster if they are vector aligned.

The following table gives the vector alignment and minimum vector length for float data:

Technology	Vector Aligment
SSE	16
AVX	32
AltiVec	16

Alignment can be controlled using a function such as `memalign`. This is a C function that is not in the ANSI standard but is available on many systems. It is defined in `malloc.h` on Linux systems.

The following macro redefines `malloc` so that all memory allocation is optimally aligned:

```
#include <malloc.h>
#define malloc(SIZE) memalign(16, SIZE)
```

Some operating systems (*e.g.* Apple's OSX) automatically align all memory to a 16-byte boundary so `memalign` is not needed.

2.3 Vector/Matrix Format

When available, vector and matrix calculations are done using single instruction, multiple data (SIMD) instructions to process several elements simultaneously. This imposes a minimum vector length given in the table below:

Technology Vector Length (floats)	minimum
SSE	4
AVX	8
AltiVec	4

For short ints (16 bits) the vector length should be twice that of the float vector length. If the vector unit supports doubles (64 bits), then the vector length should be half that of floats.

For best performance all input and output vectors should:

- have a stride of 1

  Vectors and matrices can be loaded and stored much quicker when they are contiguous in memory. The library includes special optimisations for a stride length of 2 (which was added for interleaved complex numbers), but all other non-unit strides will be significantly slower than a stride of 1 and, in many cases, almost as slow as unvectorised scalar code. Note that, a stride of -1 will also be significantly slower than a stride of $+1$.

- be vector aligned and have a vector/matrix view offset of zero or a multiple of the vector length.

  VSIPL vector and matrix views have an option to offset the start of the vector/matrix from the start of the block; for optimal performance this should be zero or a multiple of the vector length otherwise the start of the vector/matrix will not be vector aligned.

- have length greater than or equal to the vector length

  The vector unit works on arrays of the vector length so no speed up is gained by using the library on vectors of length less than this.

- have row (row major matrices) or column (column major matrices) length divisible by the vector length

  For a row major matrix: if the row length of a matrix is not divisible by the vector length then the alignment of the first element of each row will change for each row/column. For optimal performance the first element of each row should be vector aligned.

  If the row length cannot be set to a number divisible by the vector length, a matrix view can be created with a column stride divisible by vector length and greater than the row length. This technique can be used to vector align the first element of every row.

  The same rule applies to columns in column major matrices.

- have a length divisible by the vector length

Any elements at the end of the vector which cannot be dealt with by the vector unit must be dealt with in normal scalar code, which will decrease the performance. The decrease in performance becomes less important for longer vectors.

2.4 Complex Number Format

VSIPL supports two storage formats for complex numbers: split and interleaved. Which format you use depends on personal choice or the task being performed.

Split This is the default format in this implementation of VSIPL. It is what is returned when you call a create function. It is also the optimal format to use when calling most NAS VSIPL functions (see Linear Algebra section for some exceptions).

Interleaved To create data blocks in this format, you must allocate the memory yourself and then bind it to a VSIPL complex block. It is the optimal format to use when calling some linear algebra functions.

The internal storage format does not change when you admit or release real or complex data.

2.5 Error Checking and Debugging

Two versions of the NAS VSIPL library are provided: a performance version and a development version. The development version of the library (signified by a 'D' in the library's name) contains full error checking (as specified by the VSIPL standard) and should always be used when developing and debugging applications.

A few library functions return status information: always check the return code of those that do.

The performance version of the library contains no error checking, and consequently runs faster than the development library. The performance library should only be used with applications that have been run successfully with the development version of the library.

When timing code, the performance version of the library should be used.

Note: the performance version of the library reads in data before it knows how much will be used and as a result often reads more data than is needed. This is not a problem, except when using memory checkers such as Electric Fence which object to this behavior. The development library only reads in the data it intends to use and so is safe to use with memory checkers.

2.6 Support Functions

Always call `vsip_init` and `vsip_finalize` at the beginning and end of a program.

Note: For the AltiVec optimized library, calling `vsip_init` will put the AltiVec unit into non-Java mode if it is not already. This speeds up most AltiVec instructions.

See Memory Alignment and Vector/Matrix sections for full information on the optimal creation of blocks and views.

Many of the VSIPL create and bind functions have a memory hint parameter. This parameter is ignored in the current version of the library.

2.7 Scalar Functions

As the vector unit works on arrays of the vector length, scalar functions in the library are not vectorized.

2.8 Random Number Generation

The random number generation functions have not been vectorized in the current version of the library.

2.9 Vector and Elementwise Operations

All vector and elementwise operations work optimally on vectors which match the conditions given in Section 2.3. Complex vectors should be stored split (the default when using VSIPL create functions).

2.10 Signal Processing Functions

All signal processing operations work optimally on vectors which match the conditions given in Section 2.3. Complex vectors should be stored split (the default when using VSIPL create functions).

Most of the signal processing routines are split into three stages:

- a create stage
- a compute stage
- a destroy stage.

The library has been optimized to minimize the time taken to do the compute stage, which means as much precomputation as possible is done in the create stage. If you are using the same signal processing routine on many vectors of the same length, it is far quicker to just create the required signal processing object once and reuse it for each computation stage rather than recreating the object each time it is needed.

2.11 FFT Functions

To get the best performance from an FFT, a vector must have length a multiple of the numbers 2, 4, 8, and 3 only. If a vector length is not a multiple of these numbers, a DFT may be done, which is considerably slower than an FFT. Factors of 3 should be avoided if possible. An FFT will only be done for factors of 3 if the length also has a factor of 16, otherwise a DFT is done.

When doing large FFT's, optimal routines have been developed for the lengths: 4096, 8192, 16384, 32768, and 65536. These lengths should be much quicker than lengths of similar magnitude. In-place FFT's are normally faster than out-of-place FFT's. FFT's are fastest with a scale factor of 1. However, if you need to use a different scale factor, it is better to let the FFT routine do the scaling rather than to do it yourself.

The internal FFT routines only work on vector aligned data with a stride of 1. If vectors are used which do not match these restrictions an internal copy of the vector will be made. This is an important consideration when using large vectors. Also, if complex vectors are not stored split an internal copy will be made.

The current version of the NAS VSIPL library does not have any special FFT routines for doing multiple FFT's, so the time to do n single FFT's will be approximately the same as using the multiple FFT routines on a matrix of n rows.

The `ntimes` parameter to the FFT functions is ignored. The algorithmic hint is only used in the FFT create function: if the `VSIP_ALG_NOISE` hint is used, the FFT create function will take significantly longer. By default, the algorithms are optimized to minimize execution time.

2.12 FIR Filter, Convolution and Correlation Functions

These functions call the FFT functions internally and are therefore subject to the same restrictions.

Hints are ignored with the exception of the internal calling of the FFT create function described in the FFT functions section.

2.13 Linear Algebra Functions

For optimal performance the vectors and matrices used with the linear algebra functions should match the conditions given in Section 2.3. (See also the sections below when using complex LU, complex Cholesky, or complex QRD functions).

2.14 Matrix and Vector Operations

Matrix and vector operations should work optimally on row or column major matrices (row major is the default), however, the restriction exists that all matrices passed to a function should be of the same order. For example, using two row major matrices as input to a function and a column major as output will be slower than using all row major or all column major. When matrices are passed to NAS VSIPL functions that are not all of the same order, the library will assume they are all row major and treat the column major matrices as strided matrices. (See also the sections below when using LU, Cholesky, or QRD functions).

2.15 LU Decomposition, Cholesky and QRD Functions

These functions have three separate stages:

- a create stage
- a compute stage
- a destroy stage.

The library has been optimized to minimize the time taken to do the compute stage, which means as much precomputation as possible is done in the create stage. If you are using the linear algebra routine on many matrices of the same size, it is far quicker

to just create the required linear algebra object once and reuse it for each computation stage rather than recreating the object each time it is needed.

If matrices of different orders or strided matrices are passed to these functions, an internal copy will be made of the entire matrix before the computation is done. This is an important consideration when using large matrices. (Note: unaligned matrices do NOT require internal copying provided they have a stride of one and all matrices used with the functions are of the same order).

COMPLEX: unlike the rest of the NAS VSIPL library, the complex versions of these functions work on INTERLEAVED data. If non-interleaved complex matrices or vectors are passed to these functions, an interleaved internal copy will be created. This is an important consideration when using large matrices.

When using the QRD functions, it is only necessary to save the Q matrix if using the `qrdprodq` function; the `qrsol` and `qrdsolr` do not need the Q matrix.

2.16 Special Linear System Solvers

The `covsol` and `llsqsol` functions internally use the QRD functions and so have the same requirements for optimal performance, including requiring interleaved complex data.

The `toepsol` functions are based on vector operations and so have the same requirements for optimal performance.

2.17 Controlling the Number of Threads

The NAS VSIPL library is multithreaded and will take advantage of multiple cores on the processor invoking it. Utilizing multiple threads is automatic:

- The maximum number of threads used is set when `vsip_init` is called.
- The maximum number running at any one time is also set at that point. If a threaded routine is called with (say) 4 threads and we have hit this maximum number running then four of them are shut down before the new function is executed.
- The number of threads invoked when a routine is called, is decided by that routine by reference to the data (vector or matrix) size specified in the call, to provide the best performance for that call.

It is possible to change the maximum number of threads used.

1. A threaded, and a non-threaded ("serial") version of the library are provided. If you wish to only ever use one thread in a library call, use the serial version of the library.
2. The maximum number of threads used for a specific function call, and the maximum number kept running at any one time, can be changed by a call to the

routine `Thread_SetParams` with arguments `num_threads` and `max_num_running`. This call, if used, *must* be made before the library initialization routine `vsip_init` is called.

3. When calling the routine `Thread_SetParams` the value of `max_num_running` must be greater or equal to 3*`num_threads`. If the user enters a smaller value than this in their `Thread_SetParams` function call then the function will set the value of `max_num_running` to 3*`num_threads`.

   If no call to `Thread_SetParams` is made, the library default values will be utilized.

Chapter 3. Support Functions

3.1 Initialization and Finalization

- vsip_init
- vsip_finalize
- Thread_SetParams

vsip_init

Provides initialization, allowing the library to allocate and set a global state, and prepare to support the use of VSIPL functionality by the user.

Prototype

```
int vsip_init(
                    void *ptr);
```

Parameters

- ptr, pointer to structure, input.

Return Value

- Error code.

Description

This required function informs the VSIPL library that library initialization is requested, and that other VSIPL functions will be called. This function must be called at least once by all VSIPL programs. It may be called multiple times as well, with corresponding calls to vsip_finalize to create nested pairs of initialization/termination. Only the vsip_finalize matching the first vsip_init call will actually release the library. Intermediate calls to vsip_init have no effect, but support easy program/library development through compositional programming, where the user may not even know that a library itself invokes VSIPL.

The argument is reserved for future purposes. The NULL pointer should be passed for VSIPL 1.0 compliance.

Returns zero if the initialization succeeded, and non-zero otherwise.

Restrictions

This function may be called at any time during the execution of the program.

Errors

Notes

All programs must use the initialization function before calling any other VSIPL functions.

Unsuccessful initialization of the library is not an error. It is always signalled via the function's return value, and should always be checked by the application.

vsip_finalize

Provides cleanup and releases resources used by VSIPL (if the last of a nested series of calls), allowing the library to guarantee that any resources allocated by `vsip_init` are no longer in use after the call is complete.

Prototype

```
int vsip_finalize(
                    void *ptr);
```

Parameters

- `ptr`, pointer to structure, input.

Return Value

- Error code.

Description

This required function informs the VSIPL library that it is no longer being used by a program, so that all needed global state and hardware state can be returned. All programs must call this function at least once if they terminate. If the program does terminate, the last VSIPL function called must be an outermost `vsip_finalize`. Because nested `vsip_init`'s are supported, so are nested `vsip_finalize`'s.

The user must explicitly destroy all VSIPL objects before calling this function if this is an 'outermost' `vsip_finalize`. When nesting initializations, there is no need to destroy all objects prior to calling this function, but the user is obliged to keep track of the nesting depth if programs are written in such a manner.

Returns zero if the finalization succeeded, and non-zero otherwise. Zero is always returned if the call is not outermost.

Restrictions

This function may only be called if a previous `vsip_init` call has been made, with no previous corresponding `vsip_finalize`.

Errors

An outermost `vsip_finalize` call produces an error if there are any VSIPL objects not destroyed.

Notes

The user program is always responsible for returning resources it is no longer using by destroying VSIPL objects. An outermost finalization function will return resources that it allocated previously with `vsip_init`. Non-outermost `vsip_finalize`'s always return zero (success).

Thread_SetParams

Allows the library to allocate a number of threads.

Prototype

```
void Thread_SetParams(
                    vsip_length num_threads,
                    vsip_length max_num_running);
```

Parameters

- num_threads, integer scalar, input.
- max_num_running, integer scalar, input.

Return Value

- none.

Description

This function sets the maximum number of threads used for a specific function call, and the maximum number kept running.

If no call to this function is made, the library default values will be utilised.

Restrictions

This function is always included in threaded version. Calls to this function must be made before the library initialisation routine vsip_init is called.

Errors

Notes

3.2 Array and Block Functions

- vsip_Dblockadmit_P
- vsip_blockbind_P
- vsip_cblockbind_f
- vsip_Dblockcreate_P
- vsip_Dblockdestroy_P
- vsip_blockfind_P
- vsip_cblockfind_f
- vsip_blockrebind_P
- vsip_cblockrebind_f
- vsip_blockrelease_P
- vsip_cblockrelease_f
- vsip_cstorage

vsip_Dblockadmit_P

Admit a data block for VSIPL operations.

Prototype

```
int vsip_Dblockadmit_P(
                    vsip_Dblock_P   *block,
                    vsip_scalar_bl   update);
```

The following instances are supported:

```
vsip_blockadmit_f
vsip_blockadmit_i
vsip_cblockadmit_f
vsip_blockadmit_bl
vsip_blockadmit_vi
vsip_blockadmit_mi
```

Parameters

- `block`, real or complex block, length n, input.
- `update`, boolean scalar, input.

Return Value

- Error code.

Description

Admits a block of data for VSIPL operations on the associated views. Admission changes the ownership of the user data array to VSIPL, and the user should not operate on the data array after the block is admitted.

A true update flag indicates that the data in the block shall be made consistent with the user-specified data array. If the update flag is false, the block contains undefined data.

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.

Notes

It is not an error to admit a block that is already in the admitted state. A `NULL` pointer cannot be admitted.

vsip_blockbind_P

Create and bind a VSIPL block to a user-allocated data array.

Prototype

```
vsip_block_P * vsip_blockbind_P(
                    vsip_scalar_P     *user_data,
                    vsip_length        num_items,
                    vsip_memory_hint   hint);
```

The following instances are supported:

```
vsip_blockbind_f
vsip_blockbind_i
vsip_blockbind_bl
vsip_blockbind_vi
vsip_blockbind_mi
```

Parameters

- `user_data`, pointer to scalar, input.
- `num_items`, integer scalar, input.
- `hint`, enumerated type, input.

`VSIP_MEM_NONE`	no hint
`VSIP_MEM_RDONLY`	read-only
`VSIP_MEM_CONST`	constant
`VSIP_MEM_SHARED`	shared
`VSIP_MEM_SHARED_RDONLY`	shared and read-only
`VSIP_MEM_SHARED_CONST`	shared and constant

Return Value

- pointer to block.

Description

Creates a real VSIPL block object and binds the block to a user-allocated data array. The data array should contain at least `num_items` elements. The block is created in the released state and must be admitted to VSIPL before calling VSIPL functions that operate on the data.

The function returns a pointer to the block object. `NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `num_items` must be positive.
2. `hint` must be valid.

Notes

It is acceptable to bind a block to a `NULL` pointer for initialisation purposes. However, it cannot be admitted in this condition.

vsip_cblockbind_f

Create and bind a VSIPL complex block to a user-allocated data array.

Prototype

```
vsip_cblock_f * vsip_cblockbind_f(
                    vsip_scalar_f     *user_data1,
                    vsip_scalar_f     *user_data2,
                    vsip_length        num_items,
                    vsip_memory_hint   hint);
```

Parameters

- user_data1, pointer to real scalar, input. If user_data2 is NULL then user_data1 is a pointer to a data array of contiguous memory containing at least $2N$ vsip_scalar_P elements. The even elements of the data array contain the real part values, and the odd elements contain the imaginary part values. The data are stored in interleaved complex form. Note that the first element is considered to be even because index values start at zero.

  If user_data2 is not NULL then user_data1 is a pointer to a data array of contiguous memory containing at least N vsip_scalar_P elements. The data array contains the real part values. The data are stored in split complex form.

- user_data2, pointer to real scalar, input. If user_data2 is NULL then the data are stored in interleaved complex form.

  If user_data2 is not NULL then it is a pointer to a data array of contiguous memory containing at least N vsip_scalar_P elements. The data array contains the imaginary part values. The data are stored in split complex form.

- num_items, integer scalar, input.

- hint, enumerated type, input.

VSIP_MEM_NONE	no hint
VSIP_MEM_RDONLY	read-only
VSIP_MEM_CONST	constant
VSIP_MEM_SHARED	shared
VSIP_MEM_SHARED_RDONLY	shared and read-only
VSIP_MEM_SHARED_CONST	shared and constant

Return Value

- complex block.

Description

Creates a complex VSIPL block object and binds the complex block to either a single user-allocated data array containing 2`num_items` elements, or to two user-allocated data arrays each containing `num_items` elements. The block is created in the released state and must be admitted to VSIPL before calling VSIPL functions that operate on the data.

The function returns a pointer to the block object. `NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `num_items` must be positive.
2. `hint` must be valid.
3. `user_data1` must not be `NULL` if `user_data2` is not `NULL`.

Notes

It is acceptable to bind a block to a `NULL` pointer for initialization purposes. However, it cannot be admitted in this condition.

Complex data in the released state is treated as either interleaved or split as described above. A array is used for storing complex data in the interleaved format; two (identically sized) arrays, one for the real part and one for imaginary part, are used for storing complex data in the split form. The function `vsip_cstorage` will return an indicator of the desired storage format of the particular implementation. However, either storage format will work once admitted to VSIPL.

vsip_Dblockcreate_P

Creates a VSIPL block and binds a (VSIPL-allocated) data array to it.

Prototype

```
vsip_Dblock_P * vsip_Dblockcreate_P(
                    vsip_length      num_items,
                    vsip_memory_hint hint);
```

The following instances are supported:

- vsip_blockcreate_f
- vsip_blockcreate_i
- vsip_cblockcreate_f
- vsip_blockcreate_bl
- vsip_blockcreate_vi
- vsip_blockcreate_mi

Parameters

- num_items, integer scalar, input.
- hint, enumerated type, input.

VSIP_MEM_NONE	no hint
VSIP_MEM_RDONLY	read-only
VSIP_MEM_CONST	constant
VSIP_MEM_SHARED	shared
VSIP_MEM_SHARED_RDONLY	shared and read-only
VSIP_MEM_SHARED_CONST	shared and constant

Return Value

- real or complex block.

Description

Creates an admitted VSIPL block object and allocates memory for num_items elements. The size of the data array is at least num_items * sizeof(vsip_scalar_P) bytes for real data, or 2 * num_items * sizeof(vsip_scalar_P) bytes for complex data.

Data arrays created using this function can only be accessed using VSIPL functions. Information that would allow direct manipulation, such as a pointer to the data array, is not available.

The function binds the block object to the allocated data memory and returns a pointer to the block object. `NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `num_items` must be positive.
2. `hint` must be valid.

Notes

vsip_Dblockdestroy_P

Destroy a VSIPL block object and any memory allocated for it by VSIPL.

Prototype

```
void vsip_Dblockdestroy_P(
                    vsip_Dblock_P  *block);
```

The following instances are supported:

- `vsip_blockdestroy_f`
- `vsip_blockdestroy_i`
- `vsip_cblockdestroy_f`
- `vsip_blockdestroy_bl`
- `vsip_blockdestroy_vi`
- `vsip_blockdestroy_mi`

Parameters

- `block`, real or complex block, length n, input.

Return Value

- none.

Description

Destroys (frees the memory used by) a VSIPL block object.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid. It is not an error to destroy a `NULL` pointer.
2. `block` must not be derived from a complex block object.

Notes

If necessary, the programmer can determine the pointer(s) to the user-bound array(s) with a call to `vsip_blockfind_P` or `vsip_cblockfind_P` before the (released) block is destroyed.

An argument of `NULL` is not an error.

vsip_blockfind_P

Find the pointer to the data bound to a VSIPL released block object.

Prototype

```
vsip_scalar_P * vsip_blockfind_P(
                    const vsip_block_P  *block);
```

The following instances are supported:

- vsip_blockfind_f
- vsip_blockfind_i
- vsip_blockfind_bl
- vsip_blockfind_vi
- vsip_blockfind_mi

Parameters

- block, block, input.

Return Value

- pointer to scalar.

Description

Returns the address of the user data array bound to a VSIPL released block. If the block is not released, a NULL pointer is returned. NULL will also be returned is the block is bound to NULL.

Restrictions

Errors

The arguments must conform to the following:

1. block must be valid.

Notes

Although the data in a derived block is released when the parent block is released, the derived block is never in a released state so this function will fail and return `NULL`. To find the data for a derived block, the parent block must be queried.

vsip_cblockfind_f

Find the pointer(s) to the data bound to a VSIPL released complex block object.

Prototype

```
void vsip_cblockfind_f(
                    const vsip_cblock_f  *block,
                    vsip_scalar_f       **user_data1,
                    vsip_scalar_f       **user_data2);
```

Parameters

- `block`, complex block, input.
- `user_data1`, pointer to a pointer, input.
- `user_data2`, pointer to a pointer, input.

Return Value

- none.

Description

Returns the pointers to the user data array(s) bound to a VSIPL released complex block object, or `NULL`s if the complex block object data are in the admitted state.

`user_data2` will be `NULL` if the data is in complex interleaved format.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. `user_data1` and `user_data2` must not be `NULL`.

Notes

vsip_blockrebind_P

Rebind a VSIPL block to user-specified data.

Prototype

```
vsip_scalar_P * vsip_blockrebind_P(
                    vsip_block_P   *block,
                    vsip_scalar_P  *new_data);
```

The following instances are supported:

- vsip_blockrebind_f
- vsip_blockrebind_i
- vsip_blockrebind_bl
- vsip_blockrebind_vi
- vsip_blockrebind_mi

Parameters

- block, block, length n, input.
- new_data, scalar, length n, input.

Return Value

- scalar.

Description

Rebinds an existing VSIPL released real block object to a new (previously allocated) user data array. It must contain at least as many elements as the existing block object.

An attempt to rebind either a derived block object or a block object that is in an admitted state will fail: NULL will be returned in both cases. Otherwise, a pointer to the old user data array is returned.

Restrictions

Rebind does not allow you to change the number of elements in a block.

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. `new_data` must not be `NULL`.

Notes

Rebind does not allow you to change the number of elements in a block. However, there is no method to determine that the data pointer being bound is a valid pointer to an array of the proper size.

A derived block is not releasable and so may not be rebound. When the parent block is released and rebound to user data, the corresponding data in the derived block is changed.

The block must be admitted to VSIPL before calling VSIPL functions that operate on the data.

The intended use of rebind is to support efficient dynamic binding of buffers for I/O.

vsip_cblockrebind_f

Rebind a VSIPL complex block to user-specified data.

Prototype

```
void vsip_cblockrebind_f(
                     vsip_cblock_f   *block,
                     vsip_scalar_f   *new_data1,
                     vsip_scalar_f   *new_data2,
                     vsip_scalar_f  **old_data1,
                     vsip_scalar_f  **old_data2);
```

Parameters

- `block`, complex block, input.
- `new_data1`, pointer to real scalar, input. If `new_data2` is `NULL`, then `new_data1` is a pointer to a data array of contiguous memory containing at least `new_data2` pairs of `vsip_scalar_P` elements. The even elements of the data array contain the real part values, and the odd elements contain the imaginary part values. The data are stored in interleaved complex form. Note that the first element is considered to be even because index values start at zero.

  If `new_data2` is not `NULL`, then `new_data1` is a pointer to a data array of contiguous memory containing at least `new_data2` `vsip_scalar_P` elements. The data array contains the real part values. The data are stored in split complex form.
- `new_data2`, pointer to real scalar, input. If `new_data2` is `NULL`, then the data are stored in interleaved complex form.

  If `new_data2` is not `NULL`, then it is a pointer to a data array of contiguous memory containing at least `new_data2` `vsip_scalar_P` elements. The data array contains the imaginary part values. The data are stored in split complex form.
- `old_data1`, pointer to a pointer, input.
- `old_data2`, pointer to a pointer, input.

Return Value

- none.

Description

Rebinds an existing VSIPL released complex block object to either a single new (previously allocated) user-defined data array, or to two new (previously allocated) user-defined

data arrays. The new block must contain at least as many elements as the existing block object.

An attempt to rebind a block object that is in an admitted state will fail and `NULL`'s will be returned. Otherwise, a pointer to the old user data array is returned. `old_data2` will be `NULL` if the old data is in complex interleaved format.

Restrictions

Rebind does not allow you to change the number of elements in a block.

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. The pointers to the user data arrays must not be `NULL`.

Notes

Rebind does not allow you to change the number of elements in a block. However, there is no method to determine that the data pointer being bound is a valid pointer to an array of the proper size.

The block must be admitted to VSIPL before calling VSIPL functions that operate on the data.

The intended use of rebind is to support efficient dynamic binding of buffers for I/O.

vsip_blockrelease_P

Release a VSIPL block for direct user access.

Prototype

```
vsip_scalar_P * vsip_blockrelease_P(
                    vsip_block_P    *block,
                    vsip_scalar_bl   update);
```

The following instances are supported:

```
vsip_blockrelease_f
vsip_blockrelease_i
vsip_blockrelease_bl
vsip_blockrelease_vi
vsip_blockrelease_mi
```

Parameters

- `block`, block, length n, input.
- `update`, boolean scalar, input.

Return Value

- scalar.

Description

Releases a VSIPL block object to allow direct user access of the data array. Block objects created by `vsip_Dblockcreate_P` and derived blocks cannot be released: an attempt to do so will return `NULL`.

A true update flag indicates that the data in the user-specified data array shall be updated to match the data associated with the block. If the update flag is false, the state of the user data is undefined.

Returns a pointer to the data array or `NULL` if the release fails.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid.

Notes

It is not an error to release a block that is already in the released state.

If the block is derived from a complex block, only the complex block object can be released and admitted.

vsip_cblockrelease_f

Release a complex block from VSIPL for direct user access.

Prototype

```
void vsip_cblockrelease_f(
                    vsip_cblock_f    *block,
                    vsip_scalar_bl    update,
                    vsip_scalar_f   **user_data1,
                    vsip_scalar_f   **user_data2);
```

Parameters

- `block`, complex block, input.
- `update`, boolean scalar, input.
- `user_data1`, pointer to a pointer, input. If the pointer returned in `user_data2` is `NULL`, then the pointer returned in `user_data1` is a pointer to a user data array of contiguous memory containing at least $2N$ `vsip_scalar_P` elements. The even elements of the data array contain the real part values and the odd elements contain the imaginary part values. The data are stored in interleaved complex form. Note that the first element is considered to be even because index values start at zero.

  If the pointer returned in `user_data2` is not `NULL`, then the pointer returned in `user_data1` is a pointer to a user data array of contiguous memory containing at least N `vsip_scalar_P` elements, whose elements contain the real part values. The data are stored in split complex form.
- `user_data2`, pointer to a pointer, input. If the pointer returned in `user_data2` is `NULL`, then the data are stored in interleaved complex form. If the pointer returned in `user_data2` is not `NULL` then it is a pointer to a user data array of contiguous memory containing at least N `vsip_scalar_P` elements, whose elements contain the imaginary part values. The data are stored in split complex form.

Return Value

- none.

Description

Releases a VSIPL complex block object for direct user access to the data array(s). Block objects created by `vsip_Dblockcreate_P` cannot be released: an attempt to do so will return `NULL` in both pointers.

A true update flag indicates that the data in the user-specified data array shall be updated to match the data associated with the block. If the update flag is false, the state of the user data is undefined.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. The pointers to the user data arrays must not be `NULL`.

Notes

It is not an error to release a block that is already in the released state.

This function returns either a single pointer to the user data array, as the third argument (for interleaved complex data), or two pointers to the user data arrays as the third and fourth arguments (for split complex data). In the case of interleaved complex data, the fourth argument will be returned as `NULL`. If the block is not releasable, both pointers will be returned as `NULL`.

vsip_cstorage

Returns the preferred complex storage format for the system.

Prototype

```
vsip_cmplx_mem vsip_cstorage(void);
```

Parameters

- none.

Return Value

- enumerated type.

VSIP_CMPLX_INTERLEAVED	interleaved
VSIP_CMPLX_SPLIT	split
VSIP_CMPLX_NONE	no preference

Description

Returns the preferred complex storage format for the system.

Restrictions

Errors

Notes

It is also possible to determine the preferred storage format at compile time: the include file `vsip.h` defines it in the symbol `VSIP_CMPLX_MEM`.

3.3 Vector View Functions

- vsip_Dvalldestroy_P
- vsip_Dvbind_P
- vsip_Dvcloneview_P
- vsip_Dvcreate_P
- vsip_Dvdestroy_P
- vsip_Dvget_P
- vsip_Dvgetattrib_P
- vsip_Dvgetblock_P
- vsip_Dvgetlength_P
- vsip_Dvgetoffset_P
- vsip_Dvgetstride_P
- vsip_vimagview_f
- vsip_Dvput_P
- vsip_Dvputattrib_P
- vsip_Dvputlength_P
- vsip_Dvputoffset_P
- vsip_Dvputstride_P
- vsip_vrealview_f
- vsip_Dvsubview_P

vsip_Dvalldestroy_P

Destroy a vector, its associated block, and any VSIPL data array bound to the block.

Prototype

```
void vsip_Dvalldestroy_P(
                    vsip_Dvview_P  *vector);
```

The following instances are supported:

vsip_valldestroy_f
vsip_valldestroy_i
vsip_cvalldestroy_f
vsip_valldestroy_bl
vsip_valldestroy_vi
vsip_valldestroy_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- none.

Description

Destroys (frees the memory used by) a vector view object, the block object to which it is bound, and any VSIPL data array.

This function is equivalent to
vsip_Dblockdestroy_P(vsip_Dvdestroy_P(v));

This is the complementary function to vsip_Dvcreate_P and should only be used to destroy vectors that have only one view bound to the block object.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid. An argument of NULL is not an error.

2. The specified vector view must be the only view bound to the block.
3. The vector view must not be bound to a derived block.

Notes

If the vector view is bound to a block derived from a complex block, the complex block must be destroyed to free the block and associated data.

An argument of `NULL` is not an error.

vsip_Dvbind_P

Create a vector view object and bind it to a block object.

Prototype

```
vsip_Dvview_P * vsip_Dvbind_P(
                    vsip_Dblock_P  *block,
                    vsip_offset     offset,
                    vsip_stride     stride,
                    vsip_length     length);
```

The following instances are supported:

```
vsip_vbind_f
vsip_vbind_i
vsip_cvbind_f
vsip_vbind_bl
vsip_vbind_vi
vsip_vbind_mi
```

Parameters

- `block`, real or complex block, length n, input.
- `offset`, integer scalar, input.
- `stride`, integer scalar, input.
- `length`, integer scalar, input.

Return Value

- real or complex vector.

Description

Creates a vector view object or returns `NULL` if it fails. If the view create is successful, it:

1. binds the vector view object to the block object
2. sets the offset from the beginning of the data array to the beginning of the vector, the stride between scalar elements, and the length in elements (number of scalar elements)
3. returns a (pointer to the) vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. The offset `offset` must be less than the length of the block's data array.
3. The stride, length, and offset arguments must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes

It is important for the application to check the function's return value for a memory allocation failure.

vsip_Dvcloneview_P

Create a clone of a vector view.

Prototype

```
vsip_Dvview_P * vsip_Dvcloneview_P(
                    const vsip_Dvview_P  *vector);
```

The following instances are supported:

- vsip_vcloneview_f
- vsip_vcloneview_i
- vsip_cvcloneview_f
- vsip_vcloneview_bl
- vsip_vcloneview_vi
- vsip_vcloneview_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- real or complex vector.

Description

Creates a new vector view object, copies all of the attributes of the source vector view object to the new vector view object, and then binds the new vector view object to the block object of the source vector view object. This function returns NULL on a memory allocation failure; otherwise it returns a pointer to the new vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.

Notes

It is important for the application to check the return value for `NULL` in case of a memory allocation failure.

vsip_Dvcreate_P

Creates a block object and a vector view object of the block.

Prototype

```
vsip_Dvview_P * vsip_Dvcreate_P(
                    vsip_length      length,
                    vsip_memory_hint hint);
```

The following instances are supported:

- vsip_vcreate_f
- vsip_vcreate_i
- vsip_cvcreate_f
- vsip_vcreate_bl
- vsip_vcreate_vi
- vsip_vcreate_mi

Parameters

- length, integer scalar, input.
- hint, enumerated type, input.

VSIP_MEM_NONE	no hint
VSIP_MEM_RDONLY	read-only
VSIP_MEM_CONST	constant
VSIP_MEM_SHARED	shared
VSIP_MEM_SHARED_RDONLY	shared and read-only
VSIP_MEM_SHARED_CONST	shared and constant

Return Value

- real or complex vector.

Description

Creates a block object with an N element VSIPL data array, it creates a unit stride vector view object and then binds the block object to it.

This function is equivalent to
`vsip_Dvbind_P(vsip_Dblockcreate_P(N, hint), 0, 1, N);`

except that `vsip_Dvcreate_P` returns `NULL` if `vsip_Dblockcreate_P` returns `NULL`.

Restrictions

Errors

The arguments must conform to the following:

1. `length` must be positive.
2. `hint` must be valid.

Notes

vsip_Dvdestroy_P

Destroy a vector view object and return a pointer to the associated block object.

Prototype

```
vsip_Dblock_P * vsip_Dvdestroy_P(
                    vsip_Dvview_P  *vector);
```

The following instances are supported:

- vsip_vdestroy_f
- vsip_vdestroy_i
- vsip_cvdestroy_f
- vsip_vdestroy_bl
- vsip_vdestroy_vi
- vsip_vdestroy_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- pointer to real or complex block.

Description

Detaches a vector view object from the block object it was bound to, destroys (frees the memory used by) the vector view object, and returns a pointer to the block object. If the vector view argument is NULL, it returns NULL.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid. It is not an error to destroy a NULL pointer.

Notes

An argument of NULL is not an error.

vsip_Dvget_P

Get the value of a specified element of a vector view object.

Prototype

```
vsip_Dscalar_P vsip_Dvget_P(
                    const vsip_Dvview_P  *vector,
                    vsip_index            j);
```

The following instances are supported:

- vsip_vget_f
- vsip_vget_i
- vsip_cvget_f
- vsip_vget_bl
- vsip_vget_vi
- vsip_vget_mi

Parameters

- vector, real or complex vector, length n, input.
- j, vector-index scalar, input.

Return Value

- real or complex scalar.

Description

Returns the value of the specified element of a vector view, $v[j]$.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.
2. j must be a valid index of the vector view.

Notes

vsip_Dvgetattrib_P

Return the attributes of a vector view object.

Prototype

```
void vsip_Dvgetattrib_P(
                    const vsip_Dvview_P  *vector,
                    vsip_Dvattr_P        *attr);
```

The following instances are supported:

- vsip_vgetattrib_f
- vsip_vgetattrib_i
- vsip_cvgetattrib_f
- vsip_vgetattrib_bl
- vsip_vgetattrib_vi
- vsip_vgetattrib_mi

Parameters

- vector, real or complex vector, length n, input.
- attr, structure, input.
  The attribute structure contains the following information:

vsip_offset	offset	offset
vsip_stride	stride	stride
vsip_length	length	number of elements
vsip_Dblock_P *	block	data array

Return Value

- none.

Description

Returns the attributes of a vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. `vector` must be valid.
2. The attribute pointer `attr` must not be `NULL`.

Notes

The functions `vsip_Dvgetattrib_P` and `vsip_Dvputattrib_P` are not symmetric since you can get the block object but you cannot put the block object.

vsip_Dvgetblock_P

Get the block attribute of a vector view object.

Prototype

```
vsip_Dblock_P * vsip_Dvgetblock_P(
                    const vsip_Dvview_P  *vector);
```

The following instances are supported:

```
vsip_vgetblock_f
vsip_vgetblock_i
vsip_cvgetblock_f
vsip_vgetblock_bl
vsip_vgetblock_vi
vsip_vgetblock_mi
```

Parameters

- `vector`, real or complex vector, length n, input.

Return Value

- pointer to real or complex block.

Description

Returns a pointer to the VSIPL block object to which the vector view object is bound.

Restrictions

Errors

The arguments must conform to the following:

1. `vector` must be valid.

Notes

You can get the block object but you cannot put the block object.

vsip_Dvgetlength_P

Get the length attribute of a vector view object.

Prototype

```
vsip_length vsip_Dvgetlength_P(
                    const vsip_Dvview_P *vector);
```

The following instances are supported:

- vsip_vgetlength_f
- vsip_vgetlength_i
- vsip_cvgetlength_f
- vsip_vgetlength_bl
- vsip_vgetlength_vi
- vsip_vgetlength_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- integer scalar.

Description

Returns the length (number of elements) attribute of a vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.

Notes

vsip_Dvgetoffset_P

Get the offset attribute of a vector view object.

Prototype

```
vsip_offset vsip_Dvgetoffset_P(
                    const vsip_Dview_P *vector);
```

The following instances are supported:

- vsip_vgetoffset_f
- vsip_vgetoffset_i
- vsip_cvgetoffset_f
- vsip_vgetoffset_bl
- vsip_vgetoffset_vi
- vsip_vgetoffset_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- integer scalar.

Description

Returns the offset (in elements) to the first scalar element of a vector view from the start of the block object to which it is bound.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.

Notes

vsip_Dvgetstride_P

Get the stride attribute of a vector view object.

Prototype

```
vsip_stride vsip_Dvgetstride_P(
                    const vsip_Dvview_P *vector);
```

The following instances are supported:

- vsip_vgetstride_f
- vsip_vgetstride_i
- vsip_cvgetstride_f
- vsip_vgetstride_bl
- vsip_vgetstride_vi
- vsip_vgetstride_mi

Parameters

- vector, real or complex vector, length n, input.

Return Value

- integer scalar.

Description

Returns the stride (in elements of the bound block) between successive scalar elements in a vector view.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.

Notes

vsip_vimagview_f

Create a vector view object of the imaginary part of a complex vector from a complex vector view object.

Prototype

```
vsip_vview_f * vsip_vimagview_f(
                    const vsip_cvview_f  *complex_vector);
```

Parameters

- `complex_vector`, complex vector, length n, input.

Return Value

- real vector.

Description

Creates a real vector view object from the imaginary part of a complex vector view object, or returns `NULL` if it fails.

On success the function creates a derived block object (derived from the complex block object). The derived block object is bound to the imaginary part of the original complex block and then a real vector view object is bound to the derived block. The new vector encompasses the imaginary part of the input complex vector.

Restrictions

The derived block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data). Destroying the complex block is the only way to free the memory associated with the derived block object.

Errors

The arguments must conform to the following:

1. `complex_vector` must be valid.

Notes

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_vimag_P`, which copies the imaginary data (see `vsip_vimag_f`).

There are no requirements on offset or stride of a real view on its derived block. Using `vsip_Dvgetattrib_P`, information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information and the block bound to the vector to bind new vectors outside the data space of the original vector produced by `vsip_vimagview_P` will produce non-portable code. Portable code may be produced by:

1. remaining inside the data space of the vector
2. not assuming a set relationship of strides and offsets
3. using the 'get attribute' functions to obtain necessary information within the application code to understand the layout.

vsip_Dvput_P

Set the value of a specified element of a vector view object.

Prototype

```
void vsip_Dvput_P(
                    vsip_Dvview_P   *vector,
                    vsip_index       j,
                    vsip_Dscalar_P   value);
```

The following instances are supported:

- vsip_vput_f
- vsip_vput_i
- vsip_cvput_f
- vsip_vput_bl
- vsip_vput_vi
- vsip_vput_mi

Parameters

- vector, real or complex vector, length n, input.
- j, vector-index scalar, input.
- value, real or complex scalar, input.

Return Value

- none.

Description

Sets the value of the specified element of a vector view object: $v[j] := t$.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.
2. j must be a valid index of the vector view.

Notes

vsip_Dvputattrib_P

Set the attributes of a vector view object.

Prototype

```
vsip_Dvview_P * vsip_Dvputattrib_P(
                    vsip_Dvview_P       *ve3tor,
                    const vsip_Dvattr_P *attr);
```

The following instances are supported:

- vsip_vputattrib_f
- vsip_vputattrib_i
- vsip_cvputattrib_f
- vsip_vputattrib_bl
- vsip_vputattrib_vi
- vsip_vputattrib_mi

Parameters

- ve3tor, real or complex vector, length n, input.
- attr, structure, input.
  The attribute structure contains the following information:

vsip_offset	offset	offset
vsip_stride	stride	stride
vsip_length	length	number of elements
vsip_Dblock_P *	block	data array

Return Value

- real or complex vector.

Description

Sets the vector view attributes of offset, stride, and length, and returns a pointer to the vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. ve3tor must be valid.
2. The attribute pointer attr must not be NULL.
3. The stride, length, and offset arguments must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes

The functions vsip_Dvgetattrib_P and vsip_Dvputattrib_P are not symmetric since you can get the block object but you cannot put the block object.

vsip_Dvputlength_P

Set the length attribute of a vector view object.

Prototype

```
vsip_Dvview_P * vsip_Dvputlength_P(
                    vsip_Dvview_P  *vector,
                    vsip_length     length);
```

The following instances are supported:

```
vsip_vputlength_f
vsip_vputlength_i
vsip_cvputlength_f
vsip_vputlength_bl
vsip_vputlength_vi
vsip_vputlength_mi
```

Parameters

- `vector`, real or complex vector, length n, input.
- `length`, integer scalar, input.

Return Value

- real or complex vector.

Description

Sets the length (number of elements) of a vector view.

Restrictions

Errors

The arguments must conform to the following:

1. `vector` must be valid.
2. `length` must be positive.
3. The length `length` must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes

vsip_Dvputoffset_P

Set the offset attribute of a vector view object.

Prototype

```
vsip_Dvview_P * vsip_Dvputoffset_P(
                    vsip_Dvview_P  *vector,
                    vsip_offset     offset);
```

The following instances are supported:

- vsip_vputoffset_f
- vsip_vputoffset_i
- vsip_cvputoffset_f
- vsip_vputoffset_bl
- vsip_vputoffset_vi
- vsip_vputoffset_mi

Parameters

- vector, real or complex vector, length n, input.
- offset, integer scalar, input.

Return Value

- real or complex vector.

Description

Sets the offset (in elements) to the first element of a vector view, from the start of the block object's data array, to which it is bound.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.
2. The offset offset must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes

vsip_Dvputstride_P

Set the stride attribute of a vector view object.

Prototype

```
vsip_Dvview_P * vsip_Dvputstride_P(
                    vsip_Dvview_P  *vector,
                    vsip_stride     stride);
```

The following instances are supported:

- vsip_vputstride_f
- vsip_vputstride_i
- vsip_cvputstride_f
- vsip_vputstride_bl
- vsip_vputstride_vi
- vsip_vputstride_mi

Parameters

- vector, real or complex vector, length n, input.
- stride, integer scalar, input.

Return Value

- real or complex vector.

Description

Sets the stride attribute of a vector view object. Stride is the distance in elements of the block between successive elements of the vector view.

Restrictions

Errors

The arguments must conform to the following:

1. vector must be valid.
2. The stride stride must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes

vsip_vrealview_f

Create a vector view object of the real part of a complex vector from a complex vector view object.

Prototype

```
vsip_vview_f * vsip_vrealview_f(
                    const vsip_cvview_f  *complex_vector);
```

Parameters

- `complex_vector`, complex vector, length n, input.

Return Value

- real vector.

Description

Creates a real vector view object from the real part of a complex vector view object, or returns `NULL` if it fails.

On success, the function creates a derived block object (derived from the complex block object). The derived block object is bound to the real part of the original complex block and then a real vector view object is bound to the derived block. The new vector encompasses the real part of the input complex vector.

Restrictions

The derived block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data). Destroying the complex block is the only way to free the memory associated with the derived block object.

Errors

The arguments must conform to the following:

1. `complex_vector` must be valid.

Notes

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_vreal_P`, which copies the real data (see `vsip_vreal_f`).

There are no requirements on offset or stride of a real view on its derived block. Using `vsip_Dvgetattrib_P`, information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information and the block bound to the vector to bind new vectors outside the data space of the original vector produced by `vsip_vrealview_P` will produce non-portable code. Portable code may be produced by:

1. remaining inside the data space of the vector
2. not assuming a set relationship of strides and offsets
3. using the 'get attribute' functions to obtain necessary information within the application code to understand the layout.

vsip_Dvsubview_P

Create a vector view object that is a subview of a vector view object.

Prototype

```
vsip_Dvview_P * vsip_Dvsubview_P(
                    const vsip_Dvview_P  *vector,
                    vsip_index            j,
                    vsip_length           n);
```

The following instances are supported:

- vsip_vsubview_f
- vsip_vsubview_i
- vsip_cvsubview_f
- vsip_vsubview_bl
- vsip_vsubview_vi
- vsip_vsubview_mi

Parameters

- vector, real or complex vector, length n, input.
- j, vector-index scalar, input.
- n, integer scalar, input.

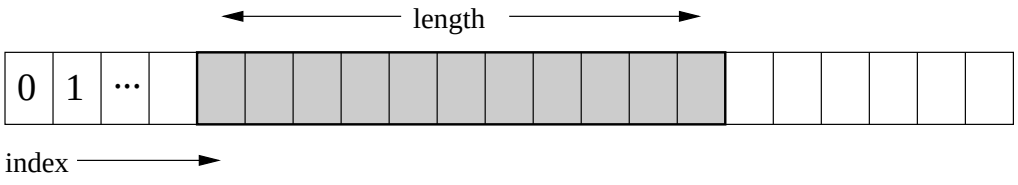
Return Value

- real or complex vector.

Description

Creates a subview vector view object from a source vector view object, and binds it to the same block object, or returns NULL if it fails. The zero index element of the new subview corresponds to the j element of the source vector view.

The subview is relative to the source view, and stride is inherited from the source view. Offset and length are relative to the source view object, not the bound block object.



Restrictions

Errors

The arguments must conform to the following:

1. `vector` must be valid.
2. `n` must be positive.
3. The subview must not extend beyond the bounds of the source view.

Notes

It is important for the application to check the return value for a memory allocation failure.

3.4 Matrix View Functions

- vsip_Dmalldestroy_P
- vsip_Dmbind_P
- vsip_Dmcloneview_P
- vsip_Dmcolview_P
- vsip_Dmcreate_P
- vsip_Dmdestroy_P
- vsip_Dmdiagview_P
- vsip_Dmget_P
- vsip_Dmgetattrib_P
- vsip_Dmgetblock_P
- vsip_Dmgetcollength_P
- vsip_Dmgetcolstride_P
- vsip_Dmgetoffset_P
- vsip_Dmgetrowlength_P
- vsip_Dmgetrowstride_P
- vsip_mimagview_f
- vsip_Dmput_P
- vsip_Dmputattrib_P
- vsip_Dmputcollength_P
- vsip_Dmputcolstride_P
- vsip_Dmputoffset_P
- vsip_Dmputrowlength_P
- vsip_Dmputrowstride_P
- vsip_mrealview_f
- vsip_Dmrowview_P
- vsip_Dmsubview_P
- vsip_Dmtransview_P

vsip_Dmalldestroy_P

Destroy a matrix, its associated block, and any VSIPL data array bound to the block.

Prototype

```
void vsip_Dmalldestroy_P(
                    vsip_Dmview_P  *matrix);
```

The following instances are supported:

```
vsip_malldestroy_f
vsip_cmalldestroy_f
```

Parameters

- `matrix`, real or complex matrix, input.

Return Value

- none.

Description

Destroys (frees the memory used by) a matrix view object, the block object to which it is bound, and any VSIPL data array.

This function is equivalent to
`vsip_Dblockdestroy_P(vsip_Dmdestroy_P(X));`

This is the complementary function to `vsip_Dmcreate_P` and should only be used to destroy matrices that have only one view bound to the block object.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid. An argument of `NULL` is not an error.
2. The specified matrix view must be the only view bound to the block.
3. The matrix view must not be bound to a derived block.

Notes

If the matrix view is bound to a block derived from a complex block, the complex block must be destroyed to free the block and associated data.

An argument of `NULL` is not an error.

vsip_Dmbind_P

Create a matrix view object and bind it to a block object.

Prototype

```
vsip_Dmview_P * vsip_Dmbind_P(
                    vsip_Dblock_P  *block,
                    vsip_offset     offset,
                    vsip_stride     col_stride,
                    vsip_length     col_length,
                    vsip_stride     row_stride,
                    vsip_length     row_length);
```

The following instances are supported:

```
vsip_mbind_f
vsip_cmbind_f
```

Parameters

- `block`, real or complex block, length n, input.
- `offset`, integer scalar, input.
- `col_stride`, integer scalar, input.
- `col_length`, integer scalar, input.
- `row_stride`, integer scalar, input.
- `row_length`, integer scalar, input.

Return Value

- real or complex matrix.

Description

Creates a matrix object or returns `NULL` if it fails. If the view create is successful, it:

1. binds the matrix view object to the block object, `block`
2. sets the offset, `offset`, from the beginning of the data array to the beginning of the matrix, the stride, `col_stride`, between scalar elements in a column, the number, `col_length`, of scalar elements in a column, the stride, `row_stride`, between scalar elements in a row, the number, `row_length`, of scalar elements in a row
3. returns a pointer to the created matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. `block` must be valid.
2. The offset `offset` must be less than the length of the block's data array.
3. The row stride, row length, column stride, column length, and offset arguments must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

It is important for the application to check the function's return value for a memory allocation failure.

vsip_Dmcloneview_P

Create a clone of a matrix view.

Prototype

```
vsip_Dmview_P * vsip_Dmcloneview_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

vsip_mcloneview_f

vsip_cmcloneview_f

Parameters

- matrix, real or complex matrix, input.

Return Value

- real or complex matrix.

Description

Creates a new matrix view object, copies all of the attributes of the source matrix view object to the new matrix view object, and then binds the new matrix view object to the block object of the source matrix view object. This function returns NULL on a memory allocation failure; otherwise it returns a pointer to the new matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

Notes

It is important for the application to check the return value for NULL in case of a memory allocation failure.

vsip_Dmcolview_P

Create a vector view object of a specified column of the source matrix view object.

Prototype

```
vsip_Dvview_P * vsip_Dmcolview_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_index            j);
```

The following instances are supported:

```
vsip_mcolview_f
vsip_cmcolview_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `j`, vector-index scalar, input.

Return Value

- real or complex vector.

Description

Creates a vector view object from a specified column of a matrix view object, or returns `NULL` if it fails. Otherwise, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified column of the source matrix object.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. `j` must be a valid column index of the source matrix view.

Notes

It is important for the application to check the return value for a memory allocation failure.

vsip_Dmcreate_P

Creates a block object and matrix view object of the block.

Prototype

```
vsip_Dmview_P * vsip_Dmcreate_P(
                    vsip_length      col_length,
                    vsip_length      row_length,
                    vsip_major       rc,
                    vsip_memory_hint hint);
```

The following instances are supported:

```
vsip_mcreate_f
vsip_cmcreate_f
```

Parameters

- `col_length`, integer scalar, input.
- `row_length`, integer scalar, input.
- `rc`, enumerated type, input.

`VSIP_ROW`	row major (C format)
`VSIP_COL`	column major (Fortran format)

- `hint`, enumerated type, input.

`VSIP_MEM_NONE`	no hint
`VSIP_MEM_RDONLY`	read-only
`VSIP_MEM_CONST`	constant
`VSIP_MEM_SHARED`	shared
`VSIP_MEM_SHARED_RDONLY`	shared and read-only
`VSIP_MEM_SHARED_CONST`	shared and constant

Return Value

- real or complex matrix.

Description

Creates a block object with an `col_length` · `row_length` element VSIPL data array, it creates an `col_length` by `row_length` dense matrix view object and then binds the block object to it.

The function
`vsip_Dmcreate_P(M, N, VSIP_ROW, hint);`
is equivalent to
`vsip_Dmbind_P(vsip_Dblockcreate_P(M*N, hint), 0, M, N, N, 1);`

and
`vsip_Dmcreate_P(M, N, VSIP_COL, hint);`
is equivalent to
`vsip_Dmbind_P(vsip_Dblockcreate_P(M*N, hint), 0, M, 1, N, M);`

except that `vsip_Dmcreate_P` returns `NULL` if `vsip_Dblockcreate_P` returns `NULL`.

Restrictions

Errors

The arguments must conform to the following:

1. `col_length` and `row_length` must be positive.
2. `rc` must be valid.
3. `hint` must be valid.

Notes

vsip_Dmdestroy_P

Destroy a matrix view object and returns a pointer to the associated block object.

Prototype

```
vsip_Dblock_P * vsip_Dmdestroy_P(
                    vsip_Dmview_P  *matrix);
```

The following instances are supported:

```
vsip_mdestroy_f
vsip_cmdestroy_f
```

Parameters

- `matrix`, real or complex matrix, input.

Return Value

- real or complex block.

Description

Detaches a matrix view object from the block object that it was bound to, destroys (frees the memory used by) the matrix view object, and returns a pointer to the block object. If the matrix view argument is `NULL`, it returns `NULL`.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid. It is not an error to destroy a `NULL` pointer.

Notes

An argument of `NULL` is not an error.

vsip_Dmdiagview_P

Create a vector view object of a matrix diagonal of a matrix view object.

Prototype

```
vsip_Dvview_P * vsip_Dmdiagview_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_stride           index);
```

The following instances are supported:

vsip_mdiagview_f

vsip_cmdiagview_f

Parameters

- matrix, real or complex matrix, input.
- index, integer scalar, input.

Return Value

- real or complex vector.

Description

Creates a vector view object of a specified diagonal of a matrix view object, or returns NULL if it fails. On success, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified diagonal of the source matrix object. An index of '0' specifies the main diagonal, positive indices are above the main diagonal, and negative indices are below the main diagonal.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.
2. index must specify a valid diagonal. For positive indices, index must be less than the number of columns; for negative indices, |index| must be less than the number of rows.

Notes

It is important for the application to check the return value for a memory allocation failure.

vsip_Dmget_P

Get the value of a specified element of a matrix view object.

Prototype

```
vsip_Dscalar_P vsip_Dmget_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_index            i,
                    vsip_index            j);
```

The following instances are supported:

vsip_mget_f

vsip_cmget_f

Parameters

- matrix, real or complex matrix, input.
- i, vector-index scalar, input.
- j, vector-index scalar, input.

Return Value

- real or complex scalar.

Description

Returns the value of the specified element of a matrix view object, $X[j, k]$.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.
2. The index must be a valid index of the matrix view.

Notes

vsip_Dmgetattrib_P

Get the attributes of a matrix view object.

Prototype

```
void vsip_Dmgetattrib_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_Dmattr_P        *attr);
```

The following instances are supported:

```
vsip_mgetattrib_f
vsip_cmgetattrib_f
```

Parameters

- matrix, real or complex matrix, input.
- attr, structure, input.
  The attribute structure contains the following information:

vsip_offset	offset	offset
vsip_stride	row_stride	row stride
vsip_stride	col_stride	column stride
vsip_length	row_length	number of columns (elements per row)
vsip_length	col_length	number of rows (elements per column)
vsip_Dblock_P *	block	data array

Return Value

- none.

Description

Returns the attributes of a matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.
2. The attribute pointer attr must not be NULL.

Notes

The functions `vsip_Dmgetattrib_P` and `vsip_Dmputattrib_P` are not symmetric since you can get the block object but you cannot put the block object.

vsip_Dmgetblock_P

Get the block attribute of a matrix view object.

Prototype

```
vsip_Dblock_P * vsip_Dmgetblock_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

```
vsip_mgetblock_f
vsip_cmgetblock_f
```

Parameters

- `matrix`, real or complex matrix, input.

Return Value

- real or complex block.

Description

Returns a pointer to the VSIPL block object to which the matrix view object is bound.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.

Notes

You can get the block object but you cannot put the block object.

vsip_Dmgetcollength_P

Get the column length attribute of a matrix view object.

Prototype

```
vsip_length vsip_Dmgetcollength_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

vsip_mgetcollength_f

vsip_cmgetcollength_f

Parameters

- matrix, real or complex matrix, input.

Return Value

- integer scalar.

Description

Returns the length of (number of elements along) a column of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

Notes

vsip_Dmgetcolstride_P

Get the column stride attribute of a matrix view object.

Prototype

```
vsip_stride vsip_Dmgetcolstride_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

vsip_mgetcolstride_f

vsip_cmgetcolstride_f

Parameters

- matrix, real or complex matrix, input.

Return Value

- integer scalar.

Description

Returns the stride (in elements of the bound block) between successive elements along a column of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

Notes

vsip_Dmgetoffset_P

Get the offset attribute of a matrix view object.

Prototype

```
vsip_offset vsip_Dmgetoffset_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

vsip_mgetoffset_f

vsip_cmgetoffset_f

Parameters

- matrix, real or complex matrix, input.

Return Value

- integer scalar.

Description

Returns the offset (in elements) to the first element of a matrix view from the start of the block object to which it is bound.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

Notes

vsip_Dmgetrowlength_P

Get the row length attribute of a matrix view object.

Prototype

```
vsip_length vsip_Dmgetrowlength_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

```
vsip_mgetrowlength_f
vsip_cmgetrowlength_f
```

Parameters

- `matrix`, real or complex matrix, input.

Return Value

- integer scalar.

Description

Returns the length of (number of elements along) a row of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.

Notes

vsip_Dmgetrowstride_P

Get the row stride attribute of a matrix view object.

Prototype

```
vsip_stride vsip_Dmgetrowstride_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

```
vsip_mgetrowstride_f
vsip_cmgetrowstride_f
```

Parameters

- `matrix`, real or complex matrix, input.

Return Value

- integer scalar.

Description

Returns the stride (in elements of the bound block) between successive elements along a row of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.

Notes

vsip_mimagview_f

Create a matrix view object of the imaginary part of complex matrix from a complex matrix view object.

Prototype

```
vsip_mview_f * vsip_mimagview_f(
                    const vsip_cmview_f  *complex_matrix);
```

Parameters

- complex_matrix, complex matrix, input.

Return Value

- real matrix.

Description

Creates a real matrix view object from the imaginary part of a complex matrix view object, or returns NULL if it fails.

On success the function creates a derived block object (derived from the complex block object). The derived block object is bound to the imaginary part of the original complex block and then a real matrix view object is bound to the derived block. The new matrix encompasses the imaginary part of the input complex matrix.

Restrictions

The derived block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data). Destroying the complex block is the only way to free the memory associated with the derived block object.

Errors

The arguments must conform to the following:

1. complex_matrix must be valid.

Notes

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_mimag_P`, which copies the imaginary data (not included in this version).

There are no requirements on offset or stride of a real view on its derived block. `vsip_Dmgetattrib_P`, information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the matrix to bind new matrixes outside the data space of the original matrix produced by `vsip_mimagview_P` will produce non-portable code. Portable code may be produced by:

1. remaining inside the data space of the matrix
2. not assuming a set relationship of strides and offsets
3. using the 'get attributes' functions to obtain necessary information within the application code to understand the layout.

vsip_Dmput_P

Set the value of a specified element of a matrix view object.

Prototype

```
void vsip_Dmput_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_index            i,
                    vsip_index            j,
                    vsip_Dscalar_P        value);
```

The following instances are supported:

vsip_mput_f

vsip_cmput_f

Parameters

- matrix, real or complex matrix, input.
- i, vector-index scalar, input.
- j, vector-index scalar, input.
- value, real or complex scalar, input.

Return Value

- none.

Description

Sets the value of the specified element of a matrix view object, $X[j, k] := t$.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.
2. The index must be a valid index of the matrix view.

Notes

vsip_Dmputattrib_P

Set the attributes of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputattrib_P(
                    vsip_Dmview_P       *matrix,
                    const vsip_Dmattr_P *attr);
```

The following instances are supported:

```
vsip_mputattrib_f
vsip_cmputattrib_f
```

Parameters

- matrix, real or complex matrix, input.
- attr, structure, input.
  The attribute structure contains the following information:

vsip_offset	offset	offset
vsip_stride	row_stride	row stride
vsip_stride	col_stride	column stride
vsip_length	row_length	number of columns (elements per row)
vsip_length	col_length	number of rows (elements per column)
vsip_Dblock_P *	block	data array

Return Value

- real or complex matrix.

Description

Sets the matrix view attributes of offset, column stride, column length, row stride, and row length, and returns a pointer to the matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

2. The attribute pointer `attr` must not be `NULL`.

3. The offset, column stride, column length, row stride, and row length arguments must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

The functions `vsip_Dmgetattrib_P` and `vsip_Dmputattrib_P` are not symmetric since you can get the block object but you cannot put the block object.

vsip_Dmputcollength_P

Set the column length attribute of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputcollength_P(
                    vsip_Dmview_P  *matrix,
                    vsip_length     n2);
```

The following instances are supported:

```
vsip_mputcollength_f
vsip_cmputcollength_f
```

Parameters

- matrix, real or complex matrix, input.
- n2, integer scalar, input.

Return Value

- real or complex matrix.

Description

Sets the length (number of elements) of a column of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.
2. n2 must be positive.
3. The length n2 must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

vsip_Dmputcolstride_P

Set the column stride attribute of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputcolstride_P(
                    vsip_Dmview_P  *matrix,
                    vsip_stride     s2);
```

The following instances are supported:

```
vsip_mputcolstride_f
vsip_cmputcolstride_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `s2`, integer scalar, input.

Return Value

- real or complex matrix.

Description

Sets the stride (in elements of the bound block) between successive elements along a column of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. The stride `s2` must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

A column stride of zero may be used to define a matrix view where each column is filled with a constant.

vsip_Dmputoffset_P

Set the offset attribute of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputoffset_P(
                    vsip_Dmview_P  *matrix,
                    vsip_offset     offset);
```

The following instances are supported:

```
vsip_mputoffset_f
vsip_cmputoffset_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `offset`, integer scalar, input.

Return Value

- real or complex matrix.

Description

Sets the offset (in elements) to the first element of a matrix view, from the start of the block, to which it is bound.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. The offset `offset` must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

vsip_Dmputrowlength_P

Set the row length attribute of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputrowlength_P(
                    vsip_Dmview_P  *matrix,
                    vsip_length     n1);
```

The following instances are supported:

```
vsip_mputrowlength_f
vsip_cmputrowlength_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `n1`, integer scalar, input.

Return Value

- real or complex matrix.

Description

Sets the length (number of elements) of a row of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. `n1` must be positive.
3. The length `n1` must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

vsip_Dmputrowstride_P

Set the row stride attribute of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmputrowstride_P(
                    vsip_Dmview_P  *matrix,
                    vsip_stride     s1);
```

The following instances are supported:

```
vsip_mputrowstride_f
vsip_cmputrowstride_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `s1`, integer scalar, input.

Return Value

- real or complex matrix.

Description

Sets the stride (in elements of the bound block) between successive elements along a row of a matrix view.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. The stride `s1` must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes

A row stride of zero may be used to define a matrix view where each row is filled with a constant.

vsip_mrealview_f

Create a matrix view object of the real part of complex matrix from a complex matrix view object.

Prototype

```
vsip_mview_f * vsip_mrealview_f(
                    const vsip_cmview_f  *complex_matrix);
```

Parameters

- complex_matrix, complex matrix, input.

Return Value

- real matrix.

Description

Creates a real matrix view object from the real part of a complex matrix view object, or returns NULL if it fails.

On success the function creates a derived block object (derived from the complex block object). The derived block object is bound to the real part of the original complex block and then a real matrix view object is bound to the derived block. The new matrix encompasses the real part of the input complex matrix.

Restrictions

The derived block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data). Destroying the complex block is the only way to free the memory associated with the derived block object.

Errors

The arguments must conform to the following:

1. complex_matrix must be valid.

Notes

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_mreal_P`, which copies the real data (not included in this version).

There are no requirements on offset or stride of a real view on its derived block. `vsip_Dmgetattrib_P`, information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the matrix to bind new matrixes outside the data space of the original matrix produced by `vsip_mrealview_P` will produce non-portable code. Portable code may be produced by:

1. remaining inside the data space of the matrix
2. not assuming a set relationship of strides and offsets
3. using the 'get attributes' functions to obtain necessary information within the application code to understand the layout.

vsip_Dmrowview_P

Create a vector view object of a specified row of the source matrix view object.

Prototype

```
vsip_Dvview_P * vsip_Dmrowview_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_index            i);
```

The following instances are supported:

```
vsip_mrowview_f
vsip_cmrowview_f
```

Parameters

- `matrix`, real or complex matrix, input.
- `i`, vector-index scalar, input.

Return Value

- real or complex vector.

Description

Creates a vector view object from a specified row of a matrix view object, or returns `NULL` if it fails. On success, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified row of the source matrix object.

Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. `i` must be a valid row index of the source matrix view.

Notes

It is important for the application to check the return value for a memory allocation failure.

vsip_Dmsubview_P

Create a matrix view object that is a subview of matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmsubview_P(
                    const vsip_Dmview_P  *matrix,
                    vsip_index            i,
                    vsip_index            j,
                    vsip_length           m,
                    vsip_length           n);
```

The following instances are supported:

vsip_msubview_f

vsip_cmsubview_f

Parameters

- matrix, real or complex matrix, input.
- i, vector-index scalar, input.
- j, vector-index scalar, input.
- m, integer scalar, input.
- n, integer scalar, input.

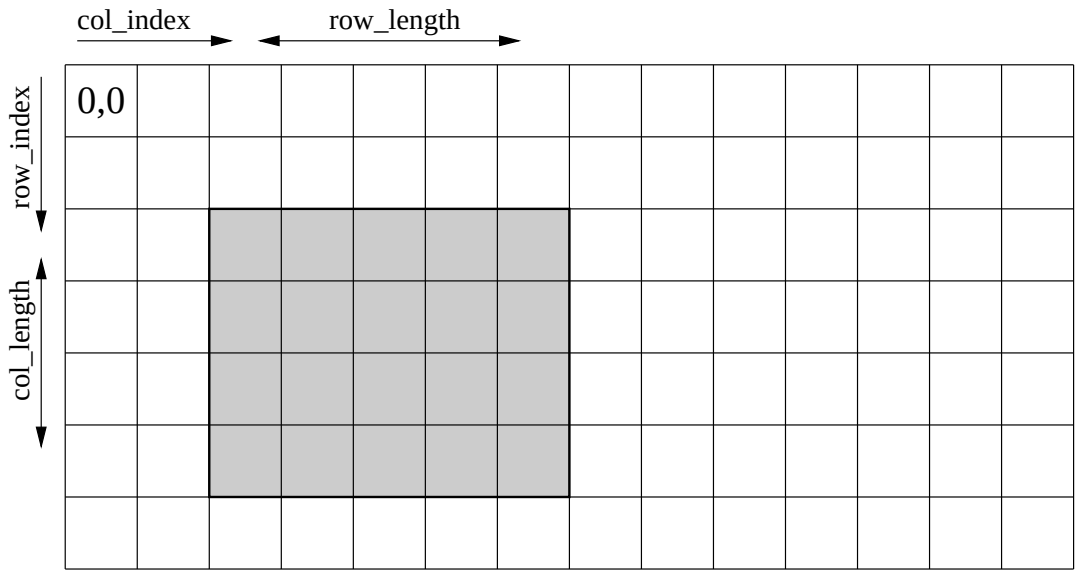
Return Value

- real or complex matrix.

Description

Creates a subview matrix view object from a source matrix view object, and binds it to the same block object, or returns NULL if it fails. The element at (0,0) in the new subview corresponds to the element at (i,j) of the source matrix view.

The subview is relative to the source view, and strides are inherited from the source view. Offset and lengths are relative to the source view object, not the bound block object.



Restrictions

Errors

The arguments must conform to the following:

1. `matrix` must be valid.
2. The matrix index (`i`,`j`) must be a valid index of the matrix view.
3. The subview must not extend beyond the bounds of the source matrix view.

Notes

It is important for the application to check the return value for a memory allocation failure.

vsip_Dmtransview_P

Create a matrix view object that is the transpose of a matrix view object.

Prototype

```
vsip_Dmview_P * vsip_Dmtransview_P(
                    const vsip_Dmview_P  *matrix);
```

The following instances are supported:

vsip_mtransview_f

vsip_cmtransview_f

Parameters

- matrix, real or complex matrix, input.

Return Value

- real or complex matrix.

Description

Creates a matrix view object that provides a transposed view of a specified a matrix view, or returns NULL if it fails. On success, it binds the new matrix view object to the same block object as the source matrix view object and sets its attributes to view the transpose of the source matrix object.

Restrictions

Errors

The arguments must conform to the following:

1. matrix must be valid.

Notes

It is important for the application to check the return value for a memory allocation failure.

Chapter 4. Scalar Functions

4.1 Complex Scalar Functions

- vsip_arg_f
- vsip_CADD_f
- vsip_cadd_f
- vsip_RCADD_f
- vsip_rcadd_f
- vsip_CDIV_f
- vsip_cdiv_f
- vsip_CRDIV_f
- vsip_crdiv_f
- vsip_CEXP_f
- vsip_cexp_f
- vsip_CJMUL_f
- vsip_cjmul_f
- vsip_cmag_f
- vsip_cmagsq_f
- vsip_CMPLX_f
- vsip_cmplx_f
- vsip_CMUL_f
- vsip_cmul_f
- vsip_RCMUL_f
- vsip_rcmul_f
- vsip_CNEG_f
- vsip_cneg_f
- vsip_CONJ_f
- vsip_conj_f
- vsip_CRECIP_f
- vsip_crecip_f
- vsip_CSQRT_f
- vsip_csqrt_f
- vsip_CSUB_f

- vsip_csub_f
- vsip_RCSUB_f
- vsip_rcsub_f
- vsip_CRSUB_f
- vsip_crsub_f
- vsip_imag_f
- vsip_polar_f
- vsip_real_f
- vsip_rect_f

vsip_arg_f

Returns the argument in radians $[-\pi, \pi]$ of a complex scalar.

Prototype

```
vsip_scalar_f vsip_arg_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- real scalar.

Description

return value := $\tan^{-1}(\text{imag}(x)/\text{real}(x))$.

Restrictions

The argument must not be zero.

Errors

Notes

vsip_CADD_f

Computes the complex sum of two scalars.

Prototype

```
void vsip_CADD_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f  y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x + y.

Restrictions

Errors

Notes

vsip_cadd_f

Computes the complex sum of two scalars.

Prototype

```
vsip_cscalar_f vsip_cadd_f(
                    vsip_cscalar_f x,
                    vsip_cscalar_f y);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x + y.

Restrictions

Errors

Notes

vsip_RCADD_f

Computes the complex sum of two scalars.

Prototype

```
void vsip_RCADD_f(
                    vsip_scalar_f    x,
                    vsip_cscalar_f   y,
                    vsip_cscalar_f  *z);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x + y.

Restrictions

Errors

Notes

vsip_rcadd_f

Computes the complex sum of two scalars.

Prototype

```
vsip_cscalar_f vsip_rcadd_f(
                    vsip_scalar_f  x,
                    vsip_cscalar_f y);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x + y.

Restrictions

Errors

Notes

vsip_CDIV_f

Computes the complex quotient of two scalars.

Prototype

```
void vsip_CDIV_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f  y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x/y.

Restrictions

The divisor must not be zero.

Errors

Notes

vsip_cdiv_f

Computes the complex quotient of two scalars.

Prototype

```
vsip_cscalar_f vsip_cdiv_f(
                    vsip_cscalar_f x,
                    vsip_cscalar_f y);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x/y.

Restrictions

The divisor must not be zero.

Errors

Notes

vsip_CRDIV_f

Computes the complex quotient of two scalars.

Prototype

```
void vsip_CRDIV_f(
                    vsip_cscalar_f  x,
                    vsip_scalar_f   y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, real scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x/y.

Restrictions

The divisor must not be zero.

Errors

Notes

vsip_crdiv_f

Computes the complex quotient of two scalars.

Prototype

```
vsip_cscalar_f vsip_crdiv_f(
                    vsip_cscalar_f x,
                    vsip_scalar_f  y);
```

Parameters

- x, complex scalar, input.
- y, real scalar, input.

Return Value

- complex scalar.

Description

return value := x/y.

Restrictions

The divisor must not be zero.

Errors

Notes

vsip_CEXP_f

Computes the exponential of a scalar.

Prototype

```
void vsip_CEXP_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f *y);
```

Parameters

- x, complex scalar, input.
- y, pointer to complex scalar, output.

Return Value

- none.

Description

*y := exp(x).

Restrictions

Overflow will occur if the argument is greater than the natural logarithm of the maximum representable number. Underflow will occur if the argument is less than the natural logarithm of the maximum representable number.

Errors

Notes

vsip_cexp_f

Computes the exponential of a scalar.

Prototype

```
vsip_cscalar_f vsip_cexp_f(
                    const vsip_cscalar_f A);
```

Parameters

- A, complex scalar, input.

Return Value

- complex scalar.

Description

return value := exp(A).

Restrictions

Overflow will occur if the argument is greater than the natural logarithm of the maximum representable number. Underflow will occur if the argument is less than the natural logarithm of the maximum representable number.

Errors

Notes

vsip_CJMUL_f

Computes the product a complex scalar with the conjugate of a second complex scalar.

Prototype

```
void vsip_CJMUL_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f  y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x · y^*.

Restrictions

Errors

Notes

vsip_cjmul_f

Computes the product a complex scalar with the conjugate of a second complex scalar.

Prototype

```
vsip_cscalar_f vsip_cjmul_f(
                    vsip_cscalar_f x,
                    vsip_cscalar_f y);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := $x \cdot y^*$.

Restrictions

Errors

Notes

vsip_cmag_f

Computes the magnitude (absolute value) of a scalar.

Prototype

```
vsip_cscalar_f vsip_cmag_f(
                    const vsip_cscalar_f A);
```

Parameters

- A, complex scalar, input.

Return Value

- complex scalar.

Description

return value := |A|.

Restrictions

Errors

Notes

vsip_cmagsq_f

Computes the magnitude squared of a complex scalar.

Prototype

```
vsip_scalar_f vsip_cmagsq_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- real scalar.

Description

return value := $|x|^2$.

Restrictions

Errors

Notes

vsip_CMPLX_f

Form a complex scalar from two real scalars.

Prototype

```
void vsip_CMPLX_f(
                    vsip_scalar_f    a,
                    vsip_scalar_f    b,
                    vsip_cscalar_f  *r);
```

Parameters

- a, real scalar, input.
- b, real scalar, input.
- r, pointer to complex scalar, output.

Return Value

- none.

Description

$*r := a + i \cdot b$.

Restrictions

Errors

Notes

vsip_cmplx_f

Form a complex scalar from two real scalars.

Prototype

```
vsip_cscalar_f vsip_cmplx_f(
                    vsip_scalar_f re,
                    vsip_scalar_f im);
```

Parameters

- re, real scalar, input.
- im, real scalar, input.

Return Value

- complex scalar.

Description

return value := re + i · im.

Restrictions

Errors

Notes

vsip_CMUL_f

Computes the complex product of two scalars.

Prototype

```
void vsip_CMUL_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f  y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x · y.

Restrictions

Errors

Notes

vsip_cmul_f

Computes the complex product of two scalars.

Prototype

```
vsip_cscalar_f vsip_cmul_f(
                    vsip_cscalar_f x,
                    vsip_cscalar_f y);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := $x \cdot y$.

Restrictions

Errors

Notes

vsip_RCMUL_f

Computes the complex product of two scalars.

Prototype

```
void vsip_RCMUL_f(
                    vsip_scalar_f    x,
                    vsip_cscalar_f   y,
                    vsip_cscalar_f  *z);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x · y.

Restrictions

Errors

Notes

vsip_rcmul_f

Computes the complex product of two scalars.

Prototype

```
vsip_cscalar_f vsip_rcmul_f(
                    vsip_scalar_f  x,
                    vsip_cscalar_f y);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := $x \cdot y$.

Restrictions

Errors

Notes

vsip_CNEG_f

Computes the negation of a complex scalar.

Prototype

```
void vsip_CNEG_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f *y);
```

Parameters

- x, complex scalar, input.
- y, pointer to complex scalar, output.

Return Value

- none.

Description

*y := −x.

Restrictions

Errors

Notes

vsip_cneg_f

Computes the negation of a complex scalar.

Prototype

```
vsip_cscalar_f vsip_cneg_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- complex scalar.

Description

return value := −x.

Restrictions

Errors

Notes

vsip_CONJ_f

Computes the complex conjugate of a scalar.

Prototype

```
void vsip_CONJ_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f *y);
```

Parameters

- x, complex scalar, input.
- y, pointer to complex scalar, output.

Return Value

- none.

Description

*y := x^*.

Restrictions

Errors

Notes

vsip_conj_f

Computes the complex conjugate of a scalar.

Prototype

```
vsip_cscalar_f vsip_conj_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x^*.

Restrictions

Errors

Notes

vsip_CRECIP_f

Computes the reciprocal of a complex scalar.

Prototype

```
void vsip_CRECIP_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f *y);
```

Parameters

- x, complex scalar, input.
- y, pointer to complex scalar, output.

Return Value

- none.

Description

*y := 1/x.

Restrictions

The argument must not be zero.

Errors

Notes

vsip_crecip_f

Computes the reciprocal of a complex scalar.

Prototype

```
vsip_cscalar_f vsip_crecip_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- complex scalar.

Description

return value := 1/x.

Restrictions

The argument must not be zero.

Errors

Notes

vsip_CSQRT_f

Computes the square root a complex scalar.

Prototype

```
void vsip_CSQRT_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f *y);
```

Parameters

- x, complex scalar, input.
- y, pointer to complex scalar, output.

Return Value

- none.

Description

*y := $\sqrt{x}$.

Restrictions

Errors

Notes

vsip_csqrt_f

Computes the square root a complex scalar.

Prototype

```
vsip_cscalar_f vsip_csqrt_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- complex scalar.

Description

return value := $\sqrt{x}$.

Restrictions

Errors

Notes

vsip_CSUB_f

Computes the complex difference of two scalars.

Prototype

```
void vsip_CSUB_f(
                    vsip_cscalar_f  x,
                    vsip_cscalar_f  y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x − y.

Restrictions

Errors

Notes

vsip_csub_f

Computes the complex difference of two scalars.

Prototype

```
vsip_cscalar_f vsip_csub_f(
                    vsip_cscalar_f x,
                    vsip_cscalar_f y);
```

Parameters

- x, complex scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x − y.

Restrictions

Errors

Notes

vsip_RCSUB_f

Computes the complex difference of two scalars.

Prototype

```
void vsip_RCSUB_f(
                    vsip_scalar_f    x,
                    vsip_cscalar_f   y,
                    vsip_cscalar_f  *z);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x − y.

Restrictions

Errors

Notes

vsip_rcsub_f

Computes the complex difference of two scalars.

Prototype

```
vsip_cscalar_f vsip_rcsub_f(
                    vsip_scalar_f  x,
                    vsip_cscalar_f y);
```

Parameters

- x, real scalar, input.
- y, complex scalar, input.

Return Value

- complex scalar.

Description

return value := x − y.

Restrictions

Errors

Notes

vsip_CRSUB_f

Computes the complex difference of two scalars.

Prototype

```
void vsip_CRSUB_f(
                    vsip_cscalar_f  x,
                    vsip_scalar_f   y,
                    vsip_cscalar_f *z);
```

Parameters

- x, complex scalar, input.
- y, real scalar, input.
- z, pointer to complex scalar, output.

Return Value

- none.

Description

*z := x − y.

Restrictions

Errors

Notes

vsip_crsub_f

Computes the complex difference of two scalars.

Prototype

```
vsip_cscalar_f vsip_crsub_f(
                    vsip_cscalar_f x,
                    vsip_scalar_f  y);
```

Parameters

- x, complex scalar, input.
- y, real scalar, input.

Return Value

- complex scalar.

Description

return value := x − y.

Restrictions

Errors

Notes

vsip_imag_f

Extract the imaginary part of a complex scalar.

Prototype

```
vsip_scalar_f vsip_imag_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- real scalar.

Description

return value := imag(x).

Restrictions

Errors

Notes

vsip_polar_f

Convert a complex scalar from rectangular to polar form. The polar data consists of a real scalar containing the radius and a corresponding real scalar containing the argument (angle) of the complex scalar.

Prototype

```
void vsip_polar_f(
                    vsip_cscalar_f  a,
                    vsip_scalar_f  *r,
                    vsip_scalar_f  *t);
```

Parameters

- a, complex scalar, input.
- r, pointer to real scalar, output.
- t, pointer to real scalar, output.

Return Value

- none.

Description

r := |a| and t := arg(a).

Restrictions

The argument must be non-zero.

Errors

Notes

Complex numbers are always stored in rectangular $x + iy$ format. The polar form is represented by two real scalars.

vsip_real_f

Extract the real part of a complex scalar.

Prototype

```
vsip_scalar_f vsip_real_f(
                    vsip_cscalar_f x);
```

Parameters

- x, complex scalar, input.

Return Value

- real scalar.

Description

return value := real(x).

Restrictions

Errors

Notes

vsip_rect_f

Convert a pair of real scalars from complex polar to complex rectangular form.

Prototype

```
vsip_cscalar_f vsip_rect_f(
                    vsip_scalar_f r,
                    vsip_scalar_f t);
```

Parameters

- r, real scalar, input.
- t, real scalar, input.

Return Value

- complex scalar.

Description

return value := $\mathtt{r} \cdot (\cos(\mathtt{t}) + \mathrm{i} \cdot \sin(\mathtt{t}))$.

Restrictions

Errors

Notes

Complex numbers are always stored in rectangular $x + \mathrm{i}y$ format. The polar form is represented by two real scalars.

4.2 Index Scalar Functions

- vsip_MATINDEX
- vsip_matindex
- vsip_mcolindex
- vsip_mrowindex

vsip_MATINDEX

Form a matrix index from two vector indices.

Prototype

```
void vsip_MATINDEX(
                    vsip_index       r,
                    vsip_index       c,
                    vsip_scalar_mi  *mi);
```

Parameters

- r, vector-index scalar, input.
- c, vector-index scalar, input.
- mi, pointer to matrix-index scalar, output.

Return Value

- none.

Description

*mi := (r, c).

Restrictions

Errors

Notes

vsip_matindex

Form a matrix index from two vector indices.

Prototype

```
vsip_scalar_mi vsip_matindex(
                     vsip_index r,
                     vsip_index c);
```

Parameters

- r, vector-index scalar, input.
- c, vector-index scalar, input.

Return Value

- matrix-index scalar.

Description

return value := (r, c).

Restrictions

Errors

Notes

vsip_mcolindex

Returns the column vector index from a matrix index.

Prototype

```
vsip_index vsip_mcolindex(
                    vsip_scalar_mi mi);
```

Parameters

- `mi`, matrix-index scalar, input.

Return Value

- vector-index scalar.

Description

return value := column(`mi`).

Restrictions

Errors

Notes

vsip_mrowindex

Returns the row vector index from a matrix index.

Prototype

```
vsip_index vsip_mrowindex(
                    vsip_scalar_mi mi);
```

Parameters

- mi, matrix-index scalar, input.

Return Value

- vector-index scalar.

Description

return value := row(mi).

Restrictions

Errors

Notes

Chapter 5. Random Number Generation

5.1 Random Number Functions

- vsip_randcreate
- vsip_randdestroy
- vsip_randu_f
- vsip_crandu_f
- vsip_vrandu_f
- vsip_cvrandu_f
- vsip_randn_f
- vsip_crandn_f
- vsip_vrandn_f
- vsip_cvrandn_f

vsip_randcreate

Create a random number generator state object.

Prototype

```
vsip_randstate * vsip_randcreate(
                     const vsip_index seed,
                     const vsip_index numprocs,
                     const vsip_index id,
                     const vsip_rng   portable);
```

Parameters

- `seed`, vector-index scalar, input. Seed to initialise the generator.
- `numprocs`, vector-index scalar, input.
- `id`, vector-index scalar, input.
- `portable`, enumerated type, input.

`VSIP_PRNG`	portable generator
`VSIP_NPRNG`	non-portable generator

Return Value

- structure.

Description

Creates a state object for use by a random number generation function. The random number generator is characterised by specifying the number of random number generators (`numprocs`) the application is expected to create, and the index (`id`) of this generator. If the portable sequence is specified, then the number of random number generators specifies how many subsequences the primary sequence is partitioned into.

The function returns a random state object which holds the state information for the random number sequence generator, or `NULL` if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. $0 < \texttt{id} \leqslant \texttt{numprocs} \leqslant 2^{31} - 1$.

Notes

You must call this function for each random number sequence/stream the application needs. This might be one per processor, one per thread, etc. For the portable sequence to have the desired pseudo-random properties, each create must specify the same number of processors/subsequences.

vsip_randdestroy

Destroys (frees the memory used by) a random number generator state object. Returns zero on success, non-zero on failure.

Prototype

```
int vsip_randdestroy(
                    vsip_randstate *rand);
```

Parameters

- rand, structure, input.

Return Value

- Error code.

Description

Destroys a random number state object.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid. An argument of NULL is not an error.

Notes

An argument of NULL is not an error.

vsip_randu_f

Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.

Prototype

```
vsip_scalar_f vsip_randu_f(
                    vsip_randstate *state);
```

Parameters

- `state`, structure, input.

Return Value

- real scalar.

Description

return value := uniform$(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The pointer to a random number state object must be valid.

Notes

vsip_crandu_f

Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.

Prototype

```
vsip_cscalar_f vsip_crandu_f(
                    vsip_randstate *state);
```

Parameters

- `state`, structure, input.

Return Value

- complex scalar.

Description

return value := $\text{uniform}(0, 1) + \text{i} \cdot \text{uniform}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The pointer to a random number state object must be valid.

Notes

The complex random number has real and imaginary components where each component is $\text{uniform}(0, 1)$.

vsip_vrandu_f

Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.

Prototype

```
void vsip_vrandu_f(
                    vsip_randstate     *state,
                    const vsip_vview_f  *R);
```

Parameters

- `state`, structure, input.
- `R`, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \text{uniform}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The pointer to a random number state object must be valid.
2. The output view object must be valid.

Notes

vsip_cvrandu_f

Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.

Prototype

```
void vsip_cvrandu_f(
                    vsip_randstate      *state,
                    const vsip_cvview_f *R);
```

Parameters

- `state`, structure, input.
- `R`, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \text{uniform}(0,1) + \mathrm{i} \cdot \text{uniform}(0,1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The pointer to a random number state object must be valid.
2. The output view object must be valid.

Notes

The complex random number has real and imaginary components where each component is uniform$(0, 1)$.

vsip_randn_f

Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Prototype

```
vsip_scalar_f vsip_randn_f(
                    vsip_randstate *state);
```

Parameters

- `state`, structure, input.

Return Value

- real scalar.

Description

return value := $\mathrm{N}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid.

Notes

If a true Gaussian random deviate is needed, the Box-Muller algorithm should be used. See Donald E. Knuth, Seminumerical Algorithms, 2nd ed., vol. 2, p117 of The Art of Computer Programming, Addison-Wesley, 1981.

vsip_crandn_f

Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Prototype

```
vsip_cscalar_f vsip_crandn_f(
                    vsip_randstate *state);
```

Parameters

- `state`, structure, input.

Return Value

- complex scalar.

Description

return value := $\mathrm{N}(0, 1) + \mathrm{i} \cdot \mathrm{N}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid.

Notes

The complex random number has real and imaginary components that are uncorrelated.

If a true Gaussian random deviate is needed, the Box-Muller algorithm should be used. See Donald E. Knuth, Seminumerical Algorithms, 2nd ed., vol. 2, p117 of The Art of Computer Programming, Addison-Wesley, 1981.

vsip_vrandn_f

Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Prototype

```
void vsip_vrandn_f(
                    vsip_randstate     *state,
                    const vsip_vview_f  *R);
```

Parameters

- `state`, structure, input.
- `R`, real vector, length n, output.

Return Value

- none.

Description

`R`$[j] := \mathrm{N}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid.
2. The output view object must be valid.

Notes

If a true Gaussian random deviate is needed, the Box-Muller algorithm should be used. See Donald E. Knuth, Seminumerical Algorithms, 2nd ed., vol. 2, p117 of The Art of Computer Programming, Addison-Wesley, 1981.

vsip_cvrandn_f

Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Prototype

```
void vsip_cvrandn_f(
                    vsip_randstate      *state,
                    const vsip_cvview_f *R);
```

Parameters

- `state`, structure, input.
- `R`, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathrm{N}(0, 1) + \mathrm{i} \cdot \mathrm{N}(0, 1)$.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid.
2. The output view object must be valid.

Notes

The complex random number has real and imaginary components that are uncorrelated.

If a true Gaussian random deviate is needed, the Box-Muller algorithm should be used. See Donald E. Knuth, Seminumerical Algorithms, 2nd ed., vol. 2, p117 of The Art of Computer Programming, Addison-Wesley, 1981.

Chapter 6. Elementwise Functions

6.1 Elementary Mathematical Functions

- vsip_vacos_f
- vsip_vasin_f
- vsip_vatan_f
- vsip_vatan2_f
- vsip_vcos_f
- vsip_vexp_f
- vsip_cvexp_f
- vsip_vexp10_f
- vsip_vlog_f
- vsip_vlog10_f
- vsip_vsin_f
- vsip_Dvsqrt_P
- vsip_vtan_f

vsip_vacos_f

Computes the principal radian value in $[0, \pi]$ of the inverse cosine for each element of a vector.

Prototype

```
void vsip_vacos_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \cos^{-1}(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

The arguments must lie in the interval $[-1, 1]$.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vasin_f

Computes the principal radian value in $[0, \pi]$ of the inverse sine for each element of a vector.

Prototype

```
void vsip_vasin_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \sin^{-1}(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

The arguments must lie in the interval $[-1, 1]$.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vatan_f

Computes the principal radian value in $[-\pi/2, \pi/2]$ of the inverse tangent for each element of a vector.

Prototype

```
void vsip_vatan_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \tan^{-1}(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vatan2_f

Computes the four-quadrant radian value in $[-\pi, \pi]$ of the inverse tangent of the ratio of the elements of two input vectors.

Prototype

```
void vsip_vatan2_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *B,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \tan^{-1}(\mathtt{A}[j]/\mathtt{B}[j])$ where $0 \leqslant j < n$.

The rules for calculating the function value are the same as those for the ANSI C function `atan2`.

Restrictions

The arguments must not be both zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vcos_f

Computes the cosine for each element of a vector. Element angle values are in radians.

Prototype

```
void vsip_vcos_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \cos(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

Accuracy is decreased for values larger than 8192.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

Input arguments are expressed in radians.

vsip_vexp_f

Computes the exponential function value for each element of a vector.

Prototype

```
void vsip_vexp_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

R$[j]$:= exp(A$[j]$) where $0 \leqslant j < n$.

Restrictions

Overflow will occur if an element is greater than the natural logarithm of the largest representable number.

Underflow will occur if an element is less than the negative of the natural logarithm of the largest representable number.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_cvexp_f

Computes the exponential function value for each element of a vector.

Prototype

```
void vsip_cvexp_f(
                    const vsip_cvview_f  *A,
                    const vsip_cvview_f  *R);
```

Parameters

- A, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

R[j] := exp(A[j]) where $0 \leqslant j < n$.

For a complex value $z = x + \mathrm{i}y$ we have $\exp(z) = \exp(x)(\cos(y) + \mathrm{i} \cdot \sin(y))$.

Restrictions

Overflow will occur if the real part of an element is greater than the natural logarithm of the largest representable number.

Underflow will occur if the real part of an element is less than the negative of the natural logarithm of the largest representable number.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vexp10_f

Computes the base 10 exponential for each element of a vector.

Prototype

```
void vsip_vexp10_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := 10^{\mathtt{A}[j]}$ where $0 \leqslant j < n$.

Restrictions

Overflow will occur if an element is greater than the base 10 logarithm of the largest representable number. Underflow will occur if an element is less than the negative of the base 10 logarithm of the largest representable number.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vlog_f

Computes the natural logarithm for each element of a vector.

Prototype

```
void vsip_vlog_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \log_e(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

Arguments must be greater than zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vlog10_f

Compute the base ten logarithm for each element of a vector.

Prototype

```
void vsip_vlog10_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \log_{10}(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

The arguments must be greater than zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vsin_f

Compute the sine for each element of a vector. Element angle values are in radians.

Prototype

```
void vsip_vsin_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \sin(\mathtt{A}[j])$ where $0 \leqslant j < n$.

Restrictions

Accuracy is decreased for values larger than 8192.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

Input arguments are expressed in radians.

vsip_Dvsqrt_P

Compute the square root for each element of a vector.

Prototype

```
void vsip_Dvsqrt_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

```
vsip_vsqrt_f
vsip_cvsqrt_f
```

Parameters

- A, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \sqrt{\mathtt{A}[j]}$ where $0 \leqslant j < n$.

Restrictions

The arguments must be greater than or equal to zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vtan_f

Compute the tangent for each element of a vector. Element angle values are in radians.

Prototype

```
void vsip_vtan_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

R$[j]$:= tan(A$[j]$) where $0 \leqslant j < n$.

Restrictions

For element values $(n + 1/2)\pi$, the tangent function has a singularity and is undefined.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

6.2 Unary Operations

- vsip_cvconj_f
- vsip_veuler_f
- vsip_Dvmag_P
- vsip_vcmagsq_f
- vsip_Dvmeanval_P
- vsip_Dvmeansqval_P
- vsip_Dvmodulate_P
- vsip_Dvneg_P
- vsip_Dvrecip_P
- vsip_vrsqrt_f
- vsip_vsq_f
- vsip_Dvsumval_P
- vsip_vsumsqval_f

vsip_cvconj_f

Compute the conjugate for each element of a complex vector.

Prototype

```
void vsip_cvconj_f(
                     const vsip_cvview_f  *A,
                     const vsip_cvview_f  *R);
```

Parameters

- A, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j]^*$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_veuler_f

Computes the complex numbers corresponding to the angle of a unit vector in the complex plane for each element of a vector.

Prototype

```
void vsip_veuler_f(
                    const vsip_vview_f  *A,
                    const vsip_cvview_f *R);
```

Parameters

- A, real vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \cos(\mathtt{A}[j]) + \mathrm{i} \cdot \sin(\mathtt{A}[j]))$ where $0 \leqslant j < n$.

Restrictions

In-place operation implies that the input is either a derived real or imaginary view of the output view.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be "in-place" as described in the restrictions, or must not overlap.

Notes

The speed may be adversely affected for large arguments because of conversion of the argument to its principal value.

vsip_Dvmag_P

Compute the magnitude for each element of a vector.

Prototype

```
void vsip_Dvmag_P(
                    const vsip_Dvview_P  *A,
                    const vsip_vview_P   *R);
```

The following instances are supported:

vsip_vmag_f

vsip_vmag_i

vsip_cvmag_f

Parameters

- A, real or complex vector, length n, input.
- R, vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := |\mathtt{A}[j]|$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. Arguments passed to the function whose data spaces overlap with different offsets or strides may cause overwriting of data before it is used.

Notes

vsip_vcmagsq_f

Computes the square of the magnitudes for each element of a vector.

Prototype

```
void vsip_vcmagsq_f(
                    const vsip_cvview_f  *A,
                    const vsip_vview_f   *R);
```

Parameters

- A, complex vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := |\mathtt{A}[j]|^2$ where $0 \leqslant j < n$.

Restrictions

For in-place functionality, the output must be either a real view or an imaginary view of the input complex vector. Output views that do not exactly match either a real view, or an imaginary view, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. Arguments passed to the function whose data space overlap with different offsets or strides may cause overwriting of data before it is used.

Notes

vsip_Dvmeanval_P

Returns the mean value of the elements of a vector.

Prototype

```
vsip_Dscalar_P vsip_Dvmeanval_P(
                    const vsip_Dview_P  *A);
```

The following instances are supported:

vsip_vmeanval_f

vsip_cvmeanval_f

Parameters

- A, real or complex vector, length n, input.

Return Value

- real or complex scalar.

Description

return value := $1/N \sum$ A$[j]$ where $0 \leqslant j < n$ where N is the number of elements.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

vsip_Dvmeansqval_P

Returns the mean magnitude squared value of the elements of a vector.

Prototype

```
vsip_scalar_P vsip_Dvmeansqval_P(
                    const vsip_Dvview_P  *A);
```

The following instances are supported:

vsip_vmeansqval_f

vsip_cvmeansqval_f

Parameters

- A, real or complex vector, length n, input.

Return Value

- scalar.

Description

return value := $1/N \sum |A[j]|^2$ where $0 \leqslant j < n$ where N is the number of elements.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

vsip_Dvmodulate_P

Computes the modulation of a real vector by a specified complex frequency.

Prototype

```
vsip_scalar_P vsip_Dvmodulate_P(
                    const vsip_Dvview_P *A,
                    const vsip_scalar_P  nu,
                    const vsip_scalar_P  phi,
                    const vsip_Dvview_P *R);
```

The following instances are supported:

vsip_vmodulate_f

vsip_cvmodulate_f

Parameters

- A, real or complex vector, length n, input.
- nu, scalar, input.
- phi, scalar, input.
- R, real or complex vector, length n, output.

Return Value

- scalar.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot (\cos(i \cdot \mathtt{nu} + \mathtt{phi}) + \mathrm{i} \cdot \sin(i \cdot \mathtt{nu} + \mathtt{phi}))$ where $0 \leqslant j < n$,

nu is the frequency in radians per index, and phi is the initial phase.

The function returns $N \cdot \mathtt{nu} + \mathtt{phi}$ where N is the length of the vector. This can be used as the initial phase in the nect call to provide a continuous modulation.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.

2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

To provide continuous filtering but processed by frames the return value can be used as the initial phase for the next frame.

vsip_Dvneg_P

Computes the negation for each element of a vector.

Prototype

```
void vsip_Dvneg_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_vneg_f

vsip_vneg_i

vsip_cvneg_f

Parameters

- A, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := -\mathtt{A}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvrecip_P

Computes the reciprocal for each element of a vector.

Prototype

```
void vsip_Dvrecip_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

```
vsip_vrecip_f
vsip_cvrecip_f
```

Parameters

- A, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$R[j] := 1/A[j]$ where $0 \leqslant j < n$.

Restrictions

The divisors must not be zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vrsqrt_f

Computes the reciprocal of the square root for each element of a vector.

Prototype

```
void vsip_vrsqrt_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := 1/\sqrt{\mathtt{A}[j]}$ where $0 \leqslant j < n$.

Restrictions

Arguments must be greater than zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vsq_f

Computes the square for each element of a vector.

Prototype

```
void vsip_vsq_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j]^2$ where $0 \leqslant j < n$.

Restrictions

Overflow will occur if an element's magnitude is greater than the square root of the largest representable number. Underflow will occur if an element's magnitude is less than the square root of the minimum smallest representable number.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvsumval_P

Returns the sum of the elements of a vector.

Prototype

```
vsip_scalar_P vsip_Dvsumval_P(
                    const vsip_vview_P  *A);
```

The following instances are supported:

```
vsip_vsumval_f
vsip_vsumval_bl
```

Parameters

- A, vector, length n, input.

Return Value

- scalar.

Description

return value $:= \sum$ A$[j]$ where $0 \leqslant j < n$.

Restrictions

Overflow may occur.

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

The order of summation is not specified, therefore significant numerical errors may occur.

vsip_vsumsqval_f

Returns the sum of the squares of the elements of a vector.

Prototype

```
vsip_scalar_f vsip_vsumsqval_f(
                    const vsip_vview_f  *A);
```

Parameters

- A, real vector, length n, input.

Return Value

- real scalar.

Description

return value := $\sum$ A$[j]^2$ where $0 \leqslant j < n$.

Restrictions

Overflow may occur.

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

The order of summation is not specified, therefore significant numerical errors may occur.

6.3 Binary Operations

- vsip_Dvadd_P
- vsip_rcvadd_f
- vsip_Dsvadd_P
- vsip_rscvadd_f
- vsip_Dvdiv_P
- vsip_svdiv_f
- vsip_rscvsub_f
- vsip_Dvexpoavg_P
- vsip_vhypot_f
- vsip_cvjmul_f
- vsip_Dvmul_P
- vsip_rcvmul_f
- vsip_Dsvmul_P
- vsip_rscvmul_f
- vsip_DvDmmul_P
- vsip_rvcmmul_f
- vsip_Dvsub_P
- vsip_crvsub_f
- vsip_rcvsub_f
- vsip_Dsvsub_P

vsip_Dvadd_P

Computes the sum, by element, of two vectors.

Prototype

```
void vsip_Dvadd_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

- vsip_vadd_f
- vsip_vadd_i
- vsip_cvadd_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] + \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rcvadd_f

Computes the sum, by element, of two vectors.

Prototype

```
void vsip_rcvadd_f(
                     const vsip_vview_f  *A,
                     const vsip_cvview_f *B,
                     const vsip_cvview_f *R);
```

Parameters

- A, real vector, length n, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] + \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dsvadd_P

Computes the sum, by element, of a scalar and a vector.

Prototype

```
void vsip_Dsvadd_P(
                    const vsip_Dscalar_P  a,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_svadd_f

vsip_svadd_i

vsip_csvadd_f

Parameters

- a, real or complex scalar, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} + \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rscvadd_f

Computes the sum, by element, of a real scalar and a complex vector.

Prototype

```
void vsip_rscvadd_f(
                    const vsip_scalar_f  a,
                    const vsip_cvview_f *B,
                    const vsip_cvview_f *R);
```

Parameters

- a, real scalar, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} + \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvdiv_P

Computes the quotient, by element, of two vectors.

Prototype

```
void vsip_Dvdiv_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_vdiv_f

vsip_cvdiv_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j]/\mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Divisors must not be zero. Overflows and underflows are possible.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_svdiv_f

Computes the quotient, by element, of a scalar and a vector.

Prototype

```
void vsip_svdiv_f(
                    const vsip_scalar_f  a,
                    const vsip_vview_f  *B,
                    const vsip_vview_f  *R);
```

Parameters

- a, real scalar, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

R$[j]$:= a/B$[j]$ where $0 \leqslant j < n$.

Restrictions

Divisors must not be zero.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rscvsub_f

Computes the difference, by element, of a real scalar and a complex vector.

Prototype

```
void vsip_rscvsub_f(
                    const vsip_scalar_f  a,
                    const vsip_cvview_f *B,
                    const vsip_cvview_f *R);
```

Parameters

- a, real scalar, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} - \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvexpoavg_P

Computes an exponential weighted average, by element, of two vectors.

Prototype

```
void vsip_Dvexpoavg_P(
                    const vsip_scalar_P  a,
                    const vsip_Dvview_P *B,
                    const vsip_Dvview_P *C);
```

The following instances are supported:

vsip_vexpoavg_f

vsip_cvexpoavg_f

Parameters

- a, scalar, input.
- B, real or complex vector, length n, input.
- C, real or complex vector, length n, modified in place.

Return Value

- none.

Description

$\mathtt{C}[j] := \mathtt{a} \cdot \mathtt{B}[j] + (1 - \mathtt{a}) \cdot \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vhypot_f

Computes the square root of the sum of squares, by element, of two input vectors.

Prototype

```
void vsip_vhypot_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \sqrt{(\mathtt{A}[j])^2 + (\mathtt{B}[j])^2}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

Intermediate overflows do not occur.

vsip_cvjmul_f

Computes the product of a complex vector with the conjugate of a second complex vector, by element.

Prototype

```
void vsip_cvjmul_f(
                     const vsip_cvview_f  *A,
                     const vsip_cvview_f  *B,
                     const vsip_cvview_f  *R);
```

Parameters

- A, complex vector, length n, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{B}[j]^*$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvmul_P

Computes the product, by element, of two vectors.

Prototype

```
void vsip_Dvmul_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_vmul_f

vsip_vmul_i

vsip_cvmul_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rcvmul_f

Computes the product, by element, of two vectors.

Prototype

```
void vsip_rcvmul_f(
                     const vsip_vview_f  *A,
                     const vsip_cvview_f *B,
                     const vsip_cvview_f *R);
```

Parameters

- A, real vector, length n, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dsvmul_P

Computes the product, by element, of a scalar and a vector.

Prototype

```
void vsip_Dsvmul_P(
                    const vsip_Dscalar_P  a,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_svmul_f

vsip_svmul_i

vsip_csvmul_f

Parameters

- a, real or complex scalar, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} \cdot \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rscvmul_f

Computes the product, by element, of a real scalar and a complex vector.

Prototype

```
void vsip_rscvmul_f(
                    const vsip_scalar_f  a,
                    const vsip_cvview_f *B,
                    const vsip_cvview_f *R);
```

Parameters

- a, real scalar, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} \cdot \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_DvDmmul_P

Computes the product, by element, of a vector and the rows or columns of a matrix.

Prototype

```
void vsip_DvDmmul_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dmview_P  *B,
                    const vsip_major      major,
                    const vsip_Dmview_P  *R);
```

The following instances are supported:

`vsip_vmmul_f`

`vsip_cvmmul_f`

Parameters

- `A`, real or complex vector, input. Length n when by rows; length m when by columns.
- `B`, real or complex matrix, size m by n, input.
- `major`, enumerated type, input.

  `VSIP_ROW` apply operation to the rows
  `VSIP_COL` apply operation to the columns
- `R`, real or complex matrix, size m by n, output.

Return Value

- none.

Description

By rows: $\mathtt{R}[j,k] := \mathtt{A}[k] \cdot \mathtt{B}[j,k]$ where $0 \leqslant j < m$ and $0 \leqslant k < n$.
By columns: $\mathtt{R}[j,k] := \mathtt{A}[j] \cdot \mathtt{B}[j,k]$ where $0 \leqslant j < m$ and $0 \leqslant k < n$.

Restrictions

Errors

The arguments must conform to the following:

1. The input and output views must be conformant.

2. All view objects must be valid.
3. The input and output matrix views must be identical views of the same block (in-place), or must not overlap. The input vector view and output vector view must not overlap.
4. `major` must be valid.

Notes

vsip_rvcmmul_f

Computes the product, by element, of a vector and the rows or columns of a matrix.

Prototype

```
void vsip_rvcmmul_f(
                    const vsip_vview_f   *A,
                    const vsip_cmview_f  *B,
                    const vsip_major      major,
                    const vsip_cmview_f  *R);
```

Parameters

- A, real vector, input. Length n when by rows; length m when by columns.
- B, complex matrix, size m by n, input.
- major, enumerated type, input.

  VSIP_ROW apply operation to the rows
  VSIP_COL apply operation to the columns
- R, complex matrix, size m by n, output.

Return Value

- none.

Description

By rows: $\mathtt{R}[j,k] := \mathtt{A}[k] \cdot \mathtt{B}[j,k]$ where $0 \leqslant j < m$ and $0 \leqslant k < n$.
By columns: $\mathtt{R}[j,k] := \mathtt{A}[j] \cdot \mathtt{B}[j,k]$ where $0 \leqslant j < m$ and $0 \leqslant k < n$.

Restrictions

Errors

The arguments must conform to the following:

1. The input and output views must be conformant.
2. All view objects must be valid.
3. The input and output matrix views must be identical views of the same block (in-place), or must not overlap. The input vector view and output vector view must not overlap.
4. major must be valid.

Notes

vsip_Dvsub_P

Computes the difference, by element, of two vectors.

Prototype

```
void vsip_Dvsub_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

- vsip_vsub_f
- vsip_vsub_i
- vsip_cvsub_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] - \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_crvsub_f

Computes the difference, by element, of two vectors.

Prototype

```
void vsip_crvsub_f(
                     const vsip_cvview_f  *A,
                     const vsip_vview_f   *B,
                     const vsip_cvview_f  *R);
```

Parameters

- A, complex vector, length n, input.
- B, real vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] - \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_rcvsub_f

Computes the difference, by element, of two vectors.

Prototype

```
void vsip_rcvsub_f(
                     const vsip_vview_f  *A,
                     const vsip_cvview_f *B,
                     const vsip_cvview_f *R);
```

Parameters

- A, real vector, length n, input.
- B, complex vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] - \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dsvsub_P

Computes the difference, by element, of a scalar and a vector.

Prototype

```
void vsip_Dsvsub_P(
                    const vsip_Dscalar_P  a,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_svsub_f

vsip_svsub_i

vsip_csvsub_f

Parameters

- a, real or complex scalar, input.
- B, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{a} - \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

To subtract a scalar from a vector just multiply the scalar by minus one and use `vsip_svadd_P`.

6.4 Ternary Operations

- vsip_Dvam_P
- vsip_Dvma_P
- vsip_Dvmsa_P
- vsip_Dvmsb_P
- vsip_Dvsam_P
- vsip_Dvsbm_P
- vsip_Dvsma_P
- vsip_Dvsmsa_P

vsip_Dvam_P

Computes the sum of two vectors and product of a third vector, by element.

Prototype

```
void vsip_Dvam_P(
                    const vsip_Dvview_P *A,
                    const vsip_Dvview_P *B,
                    const vsip_Dvview_P *C,
                    const vsip_Dvview_P *R);
```

The following instances are supported:

- vsip_vam_f
- vsip_cvam_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] + \mathtt{B}[j]) \cdot \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvma_P

Computes the product of two vectors and sum of a third vector, by element.

Prototype

```
void vsip_Dvma_P(
                    const vsip_Dvview_P *A,
                    const vsip_Dvview_P *B,
                    const vsip_Dvview_P *C,
                    const vsip_Dvview_P *R);
```

The following instances are supported:

vsip_vma_f

vsip_cvma_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{B}[j] + \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvmsa_P

Computes the product of two vectors and sum of a scalar, by element.

Prototype

```
void vsip_Dvmsa_P(
                    const vsip_Dvview_P   *A,
                    const vsip_Dvview_P   *B,
                    const vsip_Dscalar_P   c,
                    const vsip_Dvview_P   *R);
```

The following instances are supported:

`vsip_vmsa_f`

`vsip_cvmsa_f`

Parameters

- `A`, real or complex vector, length n, input.
- `B`, real or complex vector, length n, input.
- `c`, real or complex scalar, input.
- `R`, real or complex vector, length n, output.

Return Value

- none.

Description

`R`$[j] :=$ `A`$[j] \cdot$ `B`$[j] +$ `c` where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvmsb_P

Computes the product of two vectors and difference of a third vector, by element.

Prototype

```
void vsip_Dvmsb_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *C,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

- vsip_vmsb_f
- vsip_cvmsb_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{B}[j] - \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvsam_P

Computes the sum of a vector and a scalar, and product with a second vector, by element.

Prototype

```
void vsip_Dvsam_P(
                    const vsip_Dvview_P   *A,
                    const vsip_Dscalar_P   b,
                    const vsip_Dvview_P   *C,
                    const vsip_Dvview_P   *R);
```

The following instances are supported:

```
vsip_vsam_f
vsip_cvsam_f
```

Parameters

- A, real or complex vector, length n, input.
- b, real or complex scalar, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] + \mathtt{b}) \cdot \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvsbm_P

Computes the difference of two vectors, and product with a third vector, by element.

Prototype

```
void vsip_Dvsbm_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B,
                    const vsip_Dvview_P  *C,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

vsip_vsbm_f

vsip_cvsbm_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] - \mathtt{B}[j]) \cdot \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvsma_P

Computes the product of a vector and a scalar, and sum with a second vector, by element.

Prototype

```
void vsip_Dvsma_P(
                    const vsip_Dvview_P   *A,
                    const vsip_Dscalar_P   b,
                    const vsip_Dvview_P   *C,
                    const vsip_Dvview_P   *R);
```

The following instances are supported:

```
vsip_vsma_f
vsip_cvsma_f
```

Parameters

- A, real or complex vector, length n, input.
- b, real or complex scalar, input.
- C, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] \cdot \mathtt{b} + \mathtt{C}[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvsmsa_P

Computes the product of a vector and a scalar, and sum with a second scalar, by element.

Prototype

```
void vsip_Dvsmsa_P(
                    const vsip_Dvview_P   *A,
                    const vsip_Dscalar_P   b,
                    const vsip_Dscalar_P   c,
                    const vsip_Dvview_P   *R);
```

The following instances are supported:

`vsip_vsmsa_f`

`vsip_cvsmsa_f`

Parameters

- `A`, real or complex vector, length n, input.
- `b`, real or complex scalar, input.
- `c`, real or complex scalar, input.
- `R`, real or complex vector, length n, output.

Return Value

- none.

Description

`R`$[j] :=$ `A`$[j] \cdot$ `b` $+$ `c` where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

6.5 Logical Operations

- vsip_valltrue_bl
- vsip_vanytrue_bl
- vsip_vleq_f
- vsip_vlge_f
- vsip_vlgt_f
- vsip_vlle_f
- vsip_vllt_f
- vsip_vlne_f

vsip_valltrue_bl

Returns true if all the elements of a vector are true.

Prototype

```
vsip_scalar_bl vsip_valltrue_bl(
                    const vsip_vview_bl  *A);
```

Parameters

- A, boolean vector, length n, input.

Return Value

- boolean scalar.

Description

return value := (for all elements $A[j] =$ `true`) where $0 \leqslant j < n$.

Restrictions

Errors

None

Notes

vsip_vanytrue_bl

Returns true if one or more elements of a vector are true.

Prototype

```
vsip_scalar_bl vsip_vanytrue_bl(
                    const vsip_vview_bl  *A);
```

Parameters

- A, boolean vector, length n, input.

Return Value

- boolean scalar.

Description

return value := (for any element $A[j] =$ `true`) where $0 \leqslant j < n$.

Restrictions

Errors

None

Notes

The logical complement of any true is none true.

vsip_vleq_f

Computes the boolean comparison of 'equal', by element, of two vectors.

Prototype

```
void vsip_vleq_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] = \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

vsip_vlge_f

Computes the boolean comparison of 'greater than or equal', by element, of two vectors.

Prototype

```
void vsip_vlge_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *B,
                    const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] \geqslant \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

vsip_vlgt_f

Computes the boolean comparison of 'greater than', by element, of two vectors.

Prototype

```
void vsip_vlgt_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *B,
                    const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] > \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

vsip_vlle_f

Computes the boolean comparison of 'less than or equal', by element, of two vectors.

Prototype

```
void vsip_vlle_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *B,
                    const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] \leqslant \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

vsip_vllt_f

Computes the boolean comparison of 'less than', by element, of two vectors.

Prototype

```
void vsip_vllt_f(
                    const vsip_vview_f  *A,
                    const vsip_vview_f  *B,
                    const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] < \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

vsip_vlne_f

Computes the boolean comparison of 'not equal', by element, of two vectors.

Prototype

```
void vsip_vlne_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_bl *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, boolean vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := (\mathtt{A}[j] \neq \mathtt{B}[j])$ where $0 \leqslant j < n$.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes

6.6 Selection Operations

- vsip_vclip_P
- vsip_vinvclip_P
- vsip_vindexbool
- vsip_vmax_f
- vsip_vmaxmg_f
- vsip_vcmaxmgsq_f
- vsip_vcmaxmgsqval_f
- vsip_vmaxmgval_f
- vsip_vmaxval_f
- vsip_vmin_f
- vsip_vminmg_f
- vsip_vcminmgsq_f
- vsip_vcminmgsqval_f
- vsip_vminmgval_f
- vsip_vminval_f

vsip_vclip_P

Computes the generalised double clip, by element, of two vectors.

Prototype

```
void vsip_vclip_P(
                    const vsip_vview_P   *A,
                    const vsip_scalar_P   t1,
                    const vsip_scalar_P   t2,
                    const vsip_scalar_P   c1,
                    const vsip_scalar_P   c2,
                    const vsip_vview_P   *R);
```

The following instances are supported:

- vsip_vclip_f
- vsip_vclip_i

Parameters

- A, vector, length n, input.
- t1, scalar, input.
- t2, scalar, input.
- c1, scalar, input.
- c2, scalar, input.
- R, vector, length n, output.

Return Value

- none.

Description

If $A[j] \leqslant t1$ then $R[j] := c1$ else if $A[j] < t2$ then $R[j] := A[j]$ else $R[j] := c2$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

The clipping rules are evaluated sequentially: once a rule is met, the following rules are ignored. The threshold variables are unrestricted and need not be in increasing order.

vsip_vinvclip_P

Computes the generalised inverted double clip, by element, of two vectors.

Prototype

```
void vsip_vinvclip_P(
                    const vsip_vview_P  *A,
                    const vsip_scalar_P  t1,
                    const vsip_scalar_P  t2,
                    const vsip_scalar_P  t3,
                    const vsip_scalar_P  c1,
                    const vsip_scalar_P  c2,
                    const vsip_vview_P  *R);
```

The following instances are supported:

vsip_vinvclip_f

vsip_vinvclip_i

Parameters

- A, vector, length n, input.
- t1, scalar, input.
- t2, scalar, input.
- t3, scalar, input.
- c1, scalar, input.
- c2, scalar, input.
- R, vector, length n, output.

Return Value

- none.

Description

If A$[j]$ < t1 then R$[j]$:= A$[j]$ else if A$[j]$ < t2 then R$[j]$:= c1 else if A$[j]$ $\leqslant$ t3 then R$[j]$:= c2 else R$[j]$:= A$[j]$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

The clipping rules are evaluated sequentially: once a rule is met, the following rules are ignored. The threshold variables are unrestricted and need not be in increasing order.

vsip_vindexbool

Computes an index vector of the indices of the non-false elements of the boolean vector, and returns the number of non-false elements.

Prototype

```
vsip_length vsip_vindexbool(
                    const vsip_vview_bl  *X,
                    vsip_vview_vi        *Y);
```

Parameters

- X, boolean vector, length n, input.
- Y, vector-index vector, length n, output.

Return Value

- integer scalar.

Description

Returns an index vector Y of the indices of the non-false elements of the boolean vector X. The index vector is ordered: lower indices appear before higher indices. If no non-false elements are found, the index vector is unmodified, otherwise the length of the vector view is set equal to the number of non-false elements. The return value is the number of non-false elements.

Restrictions

The length of the returned index vector is dependent on the number of non-false values in the boolean object. The user must make sure that the index vector's length attribute is greater than or equal to the maximum number of non-false elements expected. If the index vector is re-used for multiple calls, its length may change after each call; therefore, the user should reset the length to the maximum value. No in-place operations are allowed.

Errors

The arguments must conform to the following:

1. All view objects must be valid.

2. The index vector must be of length greater than or equal to the number of non-false boolean elements.

Notes

VSIPL does not support zero length vectors. It is important to test the return value for zero to handle the case of no non-false elements.

vsip_vmax_f

Computes the maximum, by element, of two vectors.

Prototype

```
void vsip_vmax_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \max\{\mathtt{A}[j], \mathtt{B}[j]\}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vmaxmg_f

Computes the maximum magnitude (absolute value), by element, of two vectors.

Prototype

```
void vsip_vmaxmg_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \max\{|\mathtt{A}[j]|, |\mathtt{B}[j]|\}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vcmaxmgsq_f

Computes the maximum magnitude squared, by element, of two complex vectors.

Prototype

```
void vsip_vcmaxmgsq_f(
                    const vsip_cvview_f  *A,
                    const vsip_cvview_f  *B,
                    const vsip_vview_f   *R);
```

Parameters

- A, complex vector, length n, input.
- B, complex vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \max\{|\mathtt{A}[j]|^2, |\mathtt{B}[j]|^2\}$ where $0 \leqslant j < n$.

Restrictions

For in-place functionality, the output must be either a real view or an imaginary view of one of the input complex vectors. No in-place operation is defined for an output vector which contains both real and imaginary components of an input vector, or which does not exactly overlap a real view or an imaginary view of one of the input vectors.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vcmaxmgsqval_f

Returns the index and value of the maximum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vcmaxmgsqval_f(
                    const vsip_cvview_f  *A,
                    vsip_index           *index);
```

Parameters

- `A`, complex vector, length n, input.
- `index`, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\max\{|\mathtt{A}[j]|^2\}$ where $0 \leqslant j < n$.

If `index` is not `NULL` the index is returned.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical maximum magnitude squared values, the index of the first maximum magnitude squared is returned in the index.

vsip_vmaxmgval_f

Returns the index and value of the maximum absolute value of the elements of a vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vmaxmgval_f(
                    const vsip_vview_f  *A,
                    vsip_index          *index);
```

Parameters

- `A`, real vector, length n, input.
- `index`, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\max\{|\mathtt{A}[j]|\}$ where $0 \leqslant j < n$.

If `index` is not `NULL` the index is returned.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical maximum absolute values, the index of the first maximum absolute value is returned in the index.

vsip_vmaxval_f

Returns the index and value of the maximum value of the elements of a vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vmaxval_f(
                    const vsip_vview_f  *A,
                    vsip_index          *index);
```

Parameters

- A, real vector, length n, input.
- index, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\max\{A[j]\}$ where $0 \leqslant j < n$.

If index is not NULL the index is returned.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical maximum values the index of the first maximum is returned in the index.

vsip_vmin_f

Computes the minimum, by element, of two vectors.

Prototype

```
void vsip_vmin_f(
                      const vsip_vview_f  *A,
                      const vsip_vview_f  *B,
                      const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \min\{\mathtt{A}[j], \mathtt{B}[j]\}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vminmg_f

Computes the minimum magnitude (absolute value), by element, of two vectors.

Prototype

```
void vsip_vminmg_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_vview_f  *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \min\{|\mathtt{A}[j]|, |\mathtt{B}[j]|\}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vcminmgsq_f

Computes the minimum magnitude squared, by element, of two complex vectors.

Prototype

```
void vsip_vcminmgsq_f(
                    const vsip_cvview_f  *A,
                    const vsip_cvview_f  *B,
                    const vsip_vview_f   *R);
```

Parameters

- A, complex vector, length n, input.
- B, complex vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \min\{|\mathtt{A}[j]|^2, |\mathtt{B}[j]|^2\}$ where $0 \leqslant j < n$.

Restrictions

For in-place functionality, the output must be either a real view or an imaginary view of one of the input complex vectors. No in-place operation is defined for an output vector which contains both real and imaginary components of an input vector, or which does not exactly overlap a real view or an imaginary view of one of the input vectors.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vcminmgsqval_f

Returns the index and value of the minimum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vcminmgsqval_f(
                    const vsip_cvview_f  *A,
                    vsip_index           *index);
```

Parameters

- `A`, complex vector, length n, input.
- `index`, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\min\{|\mathtt{A}[j]|^2\}$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical minimum magnitude squared values, the index of the first minimum magnitude squared is returned in the index.

vsip_vminmgval_f

Returns the index and value of the minimum absolute value of the elements of a vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vminmgval_f(
                    const vsip_vview_f  *A,
                    vsip_index          *index);
```

Parameters

- A, real vector, length n, input.
- index, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\min\{|A[j]|\}$ where $0 \leqslant j < n$.

If index is not NULL the index is returned.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical minimum absolute value values, the index of the first minimum absolute value is returned in the index.

vsip_vminval_f

Returns the index and value of the minimum value of the elements of a vector. The index is returned by reference as one of the arguments.

Prototype

```
vsip_scalar_f vsip_vminval_f(
                    const vsip_vview_f  *A,
                    vsip_index          *index);
```

Parameters

- `A`, real vector, length n, input.
- `index`, pointer to vector-index scalar, output.

Return Value

- real scalar.

Description

return value := $\min\{A[j]\}$ where $0 \leqslant j < n$.

If `index` is not `NULL` the index is returned.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

If the vector has more than one element with identical minimum values the index of the first minimum is returned in the index.

6.7 Bitwise and Boolean Logical Operators

- vsip_vand_P
- vsip_vnot_P
- vsip_vor_P
- vsip_vxor_P

vsip_vand_P

Computes the bitwise and, by element, of two vectors.

Prototype

```
void vsip_vand_P(
                    const vsip_vview_P  *A,
                    const vsip_vview_P  *B,
                    const vsip_vview_P  *R);
```

The following instances are supported:

vsip_vand_i

vsip_vand_bl

Parameters

- A, vector, length n, input.
- B, vector, length n, input.
- R, vector, length n, output.

Return Value

- none.

Description

$R[j] := A[j]$ and $B[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vnot_P

Computes the bitwise not (one's complement), by element, of two vectors.

Prototype

```
void vsip_vnot_P(
                    const vsip_vview_P  *A,
                    const vsip_vview_P  *R);
```

The following instances are supported:

- vsip_vnot_i
- vsip_vnot_bl

Parameters

- A, vector, length n, input.
- R, vector, length n, output.

Return Value

- none.

Description

$R[j] := \text{not}(A[j])$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vor_P

Computes the bitwise inclusive or, by element, of two vectors.

Prototype

```
void vsip_vor_P(
                    const vsip_vview_P  *A,
                    const vsip_vview_P  *B,
                    const vsip_vview_P  *R);
```

The following instances are supported:

vsip_vor_i

vsip_vor_bl

Parameters

- A, vector, length n, input.
- B, vector, length n, input.
- R, vector, length n, output.

Return Value

- none.

Description

$R[j] := A[j]$ or $B[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vxor_P

Computes the bitwise exclusive or, by element, of two vectors.

Prototype

```
void vsip_vxor_P(
                    const vsip_vview_P  *A,
                    const vsip_vview_P  *B,
                    const vsip_vview_P  *R);
```

The following instances are supported:

vsip_vxor_i

vsip_vxor_bl

Parameters

- A, vector, length n, input.
- B, vector, length n, input.
- R, vector, length n, output.

Return Value

- none.

Description

R$[j]$:= A$[j]$ xor B$[j]$ where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

6.8 Element Generation and Copy

- vsip_Dvcopy_P_P
- vsip_Dmcopy_P_P
- vsip_Dvfill_P
- vsip_vramp_P

vsip_Dvcopy_P_P

Copy the source vector to the destination vector performing any necessary type conversion of the standard ANSI C scalar types.

Prototype

```
void vsip_Dvcopy_P_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *R);
```

The following instances are supported:

- vsip_vcopy_f_f
- vsip_vcopy_f_i
- vsip_vcopy_f_bl
- vsip_vcopy_i_f
- vsip_vcopy_i_i
- vsip_vcopy_i_vi
- vsip_vcopy_bl_f
- vsip_vcopy_bl_bl
- vsip_vcopy_vi_i
- vsip_vcopy_vi_vi
- vsip_vcopy_mi_mi
- vsip_cvcopy_f_f

Parameters

- A, real or complex vector, length n, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j]$ where $0 \leqslant j < n$.

Restrictions

If the source and destination overlap, the result is undefined.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), if and only if the source and destination data types are of the same size; otherwise they must not overlap.

Notes

When copying from a boolean variable, false and true map onto zero and one respectively. When copying to a boolean variable, zero maps to false and everything else maps to true.

vsip_Dmcopy_P_P

Copy the source matrix to the destination matrix performing any necessary type conversion of the standard ANSI C scalar types.

Prototype

```
void vsip_Dmcopy_P_P(
                    const vsip_Dmview_P  *A,
                    const vsip_Dmview_P  *R);
```

The following instances are supported:

vsip_mcopy_f_f

vsip_cmcopy_f_f

Parameters

- A, real or complex matrix, size m by n, input.
- R, real or complex matrix, size m by n, output.

Return Value

- none.

Description

$\mathtt{R}[j,k] := \mathtt{A}[j,k]$ where $0 \leqslant j < m$ and $0 \leqslant k < n$.

Restrictions

If the source and destination overlap, the result is undefined.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), if and only if the source and destination data types are of the same size; otherwise they must not overlap.

Notes

When copying from a boolean variable, false and true map onto zero and one respectively. When copying to a boolean variable, zero maps to false and everything else maps to true.

vsip_Dvfill_P

Fill a vector with a constant value.

Prototype

```
void vsip_Dvfill_P(
                    const vsip_Dscalar_P   a,
                    const vsip_Dvview_P   *R);
```

The following instances are supported:

```
vsip_vfill_f
vsip_vfill_i
vsip_cvfill_f
```

Parameters

- a, real or complex scalar, input.
- R, real or complex vector, length n, output.

Return Value

- none.

Description

R$[j]$:= a where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_vramp_P

Computes a vector ramp by starting at an initial value and incrementing each successive element by the ramp step size.

Prototype

```
void vsip_vramp_P(
                    const vsip_scalar_P  alpha,
                    const vsip_scalar_P  beta,
                    const vsip_vview_P   *R);
```

The following instances are supported:

- vsip_vramp_f
- vsip_vramp_i

Parameters

- alpha, scalar, input.
- beta, scalar, input.
- R, vector, length n, output.

Return Value

- none.

Description

R$[j]$:= alpha $+ j \cdot$ beta where $0 \leqslant j < n$.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes

6.9 Manipulation Operations

- vsip_vcmplx_f
- vsip_Dvgather_P
- vsip_vimag_f
- vsip_vpolar_f
- vsip_vreal_f
- vsip_vrect_f
- vsip_Dvscatter_P
- vsip_Dvswap_P

vsip_vcmplx_f

Form a complex vector from two real vectors.

Prototype

```
void vsip_vcmplx_f(
                     const vsip_vview_f  *A,
                     const vsip_vview_f  *B,
                     const vsip_cvview_f *R);
```

Parameters

- A, real vector, length n, input.
- B, real vector, length n, input.
- R, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{R}[j] := \mathtt{A}[j] + \mathrm{i} \cdot \mathtt{B}[j]$ where $0 \leqslant j < n$.

Restrictions

In-place operation for this function means the input vectors (one or both) are either a real view, or an imaginary view, of the output vector. No in-place operation is defined for an input vector which contains both real and imaginary components of the output vector, or which do not exactly overlap a real view or an imaginary view of the output vector.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvgather_P

The gather operation selects elements of a source vector using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.

Prototype

```
void vsip_Dvgather_P(
                    const vsip_Dvview_P  *X,
                    const vsip_vview_vi  *I,
                    const vsip_Dvview_P  *Y);
```

The following instances are supported:

vsip_vgather_f

vsip_vgather_i

vsip_cvgather_f

Parameters

- X, real or complex vector, length m, input.
- I, vector-index vector, length n, input.
- Y, real or complex vector, length n, output.

Return Value

- none.

Description

$Y[j] := X[I[j]]$ where $0 \leqslant j < n$.

Restrictions

The length of the destination vector must be the same size as the index vector.

Errors

The arguments must conform to the following:

1. The index and output vectors views must be the same length.

2. All view objects must be valid.
3. Index values in the index vector must be valid indexes into the source vector.

Notes

The destination vector must be the same size as the index vector.

vsip_vimag_f

Extract the imaginary part of a complex vector.

Prototype

```
void vsip_vimag_f(
                    const vsip_cvview_f  *A,
                    const vsip_vview_f   *R);
```

Parameters

- A, complex vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

R[j] := imag(A[j]) where $0 \leqslant j < n$.

Restrictions

If done in-place the output is placed in a real or imaginary view of the input. No in-place functionality is defined which places the output in a view which encompasses both real and imaginary space in the input vector. The output vector for in-place must exactly overlap the data space of the real view or the imaginary view of the input, and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

Frequently it would be preferable to use the support function `vsip_imagview_p` instead of `vsip_imag_p`. The difference is whether a copy of the imaginary portion of the vector is made, or just a view of the imaginary portion is returned.

vsip_vpolar_f

Convert a complex vector from rectangular to polar form. The polar data consists of a real vector containing the radius and a corresponding real vector containing the argument (angle) of the complex input data.

Prototype

```
void vsip_vpolar_f(
                    const vsip_cvview_f  *A,
                    const vsip_vview_f   *R,
                    const vsip_vview_f   *P);
```

Parameters

- A, complex vector, length n, input.
- R, real vector, length n, output.
- P, real vector, length n, output.

Return Value

- none.

Description

R$[j] := |$A$[j]|$ and P$[j] := \arg($A$[j])$ where $0 \leqslant j < n$.

Restrictions

In-place operation for this function requires that the radius and argument output vectors be placed in a real or imaginary view of the input vector. No in-place functionality is defined where an output view contains both real and imaginary data space. The in-place real or imaginary view must exactly overlap the input data space and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

For in-place there is no requirement on which view which output vector is placed in. So the radius vector could go in either the real or imaginary view, and the argument vector would go in the view not used by the radius vector. Complex numbers are always in rectangular format. The polar form is represented by two real vectors.

vsip_vreal_f

Extract the real part of a complex vector.

Prototype

```
void vsip_vreal_f(
                    const vsip_cvview_f  *A,
                    const vsip_vview_f   *R);
```

Parameters

- A, complex vector, length n, input.
- R, real vector, length n, output.

Return Value

- none.

Description

$R[j] := \text{real}(A[j])$ where $0 \leqslant j < n$.

Restrictions

If done in-place the output is placed in a real or imaginary view of the input. No in-place functionality is defined which places the output in a view which encompasses both real and imaginary space in the input vector. The output vector for in-place must exactly overlap the data space of the real view or the imaginary view of the input, and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

The support function `vsip_srealview_p` will often be preferable to `vsip_sreal_d`. The difference is whether a copy of the real portion of the vector is made, or just a view of the real portion is returned.

vsip_vrect_f

Convert a pair of real vectors from complex polar to complex rectangular form.

Prototype

```
void vsip_vrect_f(
                    const vsip_vview_f  *R,
                    const vsip_vview_f  *P,
                    const vsip_cvview_f *A);
```

Parameters

- R, real vector, length n, input.
- P, real vector, length n, input.
- A, complex vector, length n, output.

Return Value

- none.

Description

$\mathtt{A}[j] := \mathtt{R}[j] \cdot (\cos(\mathtt{P}[j]) + \mathrm{i} \cdot \sin(\mathtt{P}[j]))$ where $0 \leqslant j < n$.

Restrictions

In-place operation for this function requires that the radius and argument input vectors be in a real or imaginary view of the output vector. No in-place functionality is defined where an input view contains both real and imaginary data space of the output view. For in-place the data in the views must exactly overlap and not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

For in-place either the real or imaginary view of the output can hold the radius data and the other view holds the argument data. Complex numbers are always in rectangular format. The polar form is represented by two real vectors.

vsip_Dvscatter_P

The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector index is used to select a storage location in the output vector to store the element from the source vector.

Prototype

```
void vsip_Dvscatter_P(
                    const vsip_Dvview_P  *X,
                    const vsip_Dvview_P  *Y,
                    const vsip_vview_vi  *I);
```

The following instances are supported:

- `vsip_vscatter_f`
- `vsip_vscatter_i`
- `vsip_cvscatter_f`

Parameters

- `X`, real or complex vector, length n, input.
- `Y`, real or complex vector, length m, output.
- `I`, vector-index vector, length n, input.

Return Value

- none.

Description

`Y`$[$`I`$[j]] :=$ `X`$[j]$ where $0 \leqslant j < n$.

Restrictions

If the index vector contains duplicate entries, the value stored in the destination will be from the source vector, but which one is undefined. There is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. The index and input vectors must have identical lengths.
2. All view objects must be valid.
3. Index values in the index vector must be valid indexes into the output.

Notes

The view of the destination vector is not modified. Values in the destination not indexed are not modified.

vsip_Dvswap_P

Swap elements between two vectors.

Prototype

```
void vsip_Dvswap_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B);
```

The following instances are supported:

vsip_vswap_f

vsip_cvswap_f

Parameters

- A, real or complex vector, length n, modified in place.
- B, real or complex vector, length n, modified in place.

Return Value

- none.

Description

$t := \mathtt{A}[j]$
$\mathtt{A}[j] := \mathtt{B}[j]$
$\mathtt{B}[j] := t$
where $0 \leqslant j < n$.

Restrictions

This function may not be done in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must not overlap.

Notes

Chapter 7. Signal Processing Functions

7.1 FFT Functions

- vsip_ccfftip_create_f
- vsip_ccfftop_create_f
- vsip_crfftop_create_f
- vsip_rcfftop_create_f
- vsip_fft_destroy_f
- vsip_fft_getattr_f
- vsip_ccfftip_f
- vsip_ccfftop_f
- vsip_crfftop_f
- vsip_rcfftop_f
- vsip_ccfftmip_create_f
- vsip_ccfftmop_create_f
- vsip_crfftmop_create_f
- vsip_rcfftmop_create_f
- vsip_fftm_destroy_f
- vsip_fftm_getattr_f
- vsip_ccfftmip_f
- vsip_ccfftmop_f
- vsip_crfftmop_f
- vsip_rcfftmop_f

vsip_ccfftip_create_f

Create a 1D FFT object.

Prototype

```
vsip_fft_f * vsip_ccfftip_create_f(
                    const vsip_index    length,
                    const vsip_scalar_f scale,
                    const vsip_fft_dir  dir,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `length`, vector-index scalar, input. The length N of the data vector.
- `scale`, real scalar, input. Typical scale factors are 1, $1/N$ and $1/\sqrt{N}$.
- `dir`, enumerated type, input.

`VSIP_FFT_FWD`	forward
`VSIP_FFT_INV`	reverse (or inverse)

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D FFT object holding the information on the type of FFT to be computed: complex-to-complex in-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$.

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `length`, the length of the FFT, must be positive and non-zero.
2. `dir` must be valid.
3. `hint` must be valid.

Notes

FFT operations are supported on vectors of any length.

vsip_ccfftop_create_f

Create a 1D FFT object.

Prototype

```
vsip_fft_f * vsip_ccfftop_create_f(
                    const vsip_index    length,
                    const vsip_scalar_f scale,
                    const vsip_fft_dir  dir,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `length`, vector-index scalar, input. The length N of the data vector.
- `scale`, real scalar, input. Typical scale factors are 1, $1/N$ and $1/\sqrt{N}$.
- `dir`, enumerated type, input.

`VSIP_FFT_FWD`	forward
`VSIP_FFT_INV`	reverse (or inverse)

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D FFT object holding the information on the type of FFT to be computed: complex-to-complex out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$.

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `length`, the length of the FFT, must be positive and non-zero.
2. `dir` must be valid.
3. `hint` must be valid.

Notes

FFT operations are supported on vectors of any length.

vsip_crfftop_create_f

Create a 1D FFT object.

Prototype

```
vsip_fft_f * vsip_crfftop_create_f(
                    const vsip_index    length,
                    const vsip_scalar_f scale,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `length`, vector-index scalar, input. The length N of the data vector.
- `scale`, real scalar, input. Typical scale factors are 1, $1/N$ and $1/\sqrt{N}$.
- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D FFT object holding the information on the type of FFT to be computed: (reverse) complex-to-real out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$.

`NULL` is returned if the create fails.

Restrictions

The length N must be even.

Errors

The arguments must conform to the following:

1. `length`, the length of the FFT, must be positive, even and non-zero.
2. `hint` must be valid.

Notes

FFT operations are supported on vectors of any length.

vsip_rcfftop_create_f

Create a 1D FFT object.

Prototype

```
vsip_fft_f * vsip_rcfftop_create_f(
                    const vsip_index    length,
                    const vsip_scalar_f scale,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `length`, vector-index scalar, input. The length N of the data vector.
- `scale`, real scalar, input. Typical scale factors are 1, $1/N$ and $1/\sqrt{N}$.
- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D FFT object holding the information on the type of FFT to be computed: (forward) real-to-complex out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$.

`NULL` is returned if the create fails.

Restrictions

The length N must be even.

Errors

The arguments must conform to the following:

1. `length`, the length of the FFT, must be positive, even and non-zero.
2. `hint` must be valid.

Notes

FFT operations are supported on vectors of any length.

vsip_fft_destroy_f

Destroy an FFT object.

Prototype

```
int vsip_fft_destroy_f(
                    vsip_fft_f  *plan);
```

Parameters

- `plan`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) an FFT object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The input object must conform to the following:

1. The FFT object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_fft_getattr_f

Return the attributes of an FFT object.

Prototype

```
void vsip_fft_getattr_f(
                    const vsip_fft_f  *plan,
                    vsip_fft_attr_f   *attr);
```

Parameters

- plan, structure, input.
- attr, pointer to structure, output.
  The attribute structure contains the following information:

vsip_scalar_vi	input	input length
vsip_scalar_vi	output	output length
vsip_fft_place	place	in-place or out-of-place
vsip_scalar_f	scale	scale factor
vsip_fft_dir	dir	forward or reverse

Return Value

- none.

Description

Returns the attributes of an FFT object.

Restrictions

Errors

The arguments must conform to the following:

1. The FFT object plan must be valid.
2. The attribute pointer attr must not be NULL.

Notes

There is no attribute that explicitly indicates complex-to-complex, real-to-complex, or complex-to-real FFTs. This may be inferred by examining the input and output sizes.

vsip_ccfftip_f

Apply a complex-to-complex Fast Fourier Transform (FFT).

Prototype

```
void vsip_ccfftip_f(
                    const vsip_fft_f    *plan,
                    const vsip_cvview_f *xy);
```

Parameters

- `plan`, structure, input.
- `xy`, complex vector, modified in place.

Return Value

- none.

Description

Computes a complex-to-complex in-place Fast Fourier Transform (FFT) of the complex vector x, and stores the results in the complex vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$
where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$ and $sign$ is -1 for a forward transform and $+1$ for a reverse transform.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex in-place FFT object.
3. The input must be a complex vector of length N, conformant to the FFT object.

Notes

vsip_ccfftop_f

Apply a complex-to-complex Fast Fourier Transform (FFT).

Prototype

```
void vsip_ccfftop_f(
                    const vsip_fft_f    *plan,
                    const vsip_cvview_f *x,
                    const vsip_cvview_f *y);
```

Parameters

- plan, structure, input.
- x, complex vector, input.
- y, complex vector, output.

Return Value

- none.

Description

Computes a complex-to-complex out-of-place Fast Fourier Transform (FFT) of the complex vector x, and stores the results in the complex vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$
where $W_N = \exp(sign \cdot 2\pi \mathrm{i}/N)$ and $sign$ is -1 for a forward transform and $+1$ for a reverse transform.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex out-of-place FFT object.
3. The input and output must be complex vectors of length N, conformant to the FFT object.
4. The input and output vectors must not overlap.

Notes

vsip_crfftop_f

Apply a complex-to-real Fast Fourier Transform (FFT).

Prototype

```
void vsip_crfftop_f(
                    const vsip_fft_f    *plan,
                    const vsip_cvview_f *x,
                    const vsip_vview_f  *y);
```

Parameters

- plan, structure, input.
- x, complex vector, input.
- y, real vector, output.

Return Value

- none.

Description

Computes a complex-to-real out-of-place (reverse) Fast Fourier Transform (FFT) of the complex vector x, and stores the results in the real vector y.
$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(+2\pi \mathrm{i}/N)$.

Restrictions

Only unit stride vectors are supported. The length, N, must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real out-of-place FFT object.
3. The input must be a complex vector of length $N/2 + 1$, conformant to the FFT object.
4. The output must be a real vector of even length N, conformant to the FFT object.

5. The input and output vectors must not overlap.

6. The input and output vectors must be unit stride.

Notes

Generally, the FFT transforms a complex sequence into a complex sequence. However, in certain applications we may know the output sequence is real. Often, this is the case because the complex input sequence was the transform of a real sequence. In this case, you can save about half of the computational work.

For the output sequence, y, to be a real sequence, the following identity on the input sequence, x, must be true: $x_j = x^*_{N-j}$ for $\lfloor N/2 \rfloor < j < N$.

The input values x_j for $j > \lfloor N/2 \rfloor$ need not be supplied; they can be inferred from the first half of the input.

Thus, in the complex-to-real routine, x is a complex vector of length $\lfloor N/2 \rfloor + 1$ and y is a real vector of length N. Even though only $\lfloor N/2 \rfloor + 1$ input complex values are supplied, the size of the transform is still N in this case, because implicitly you are using the FFT formula for a sequence of length N.

The first value of the input vector, $x[0]$ must be a real number (that is, it must have zero imaginary part). The first value corresponds to the zero (DC) frequency component of the data. Since we restrict N to be an even number, the last value of the input vector, $x[N/2]$, must also be real. The last value corresponds to one half the Nyquist rate (or sample rate). This value is sometimes called the folding frequency. The routine assumes that these values are real; if you specify a non-zero imaginary part, it is ignored.

vsip_rcfftop_f

Apply a real-to-complex Fast Fourier Transform (FFT).

Prototype

```
void vsip_rcfftop_f(
                    const vsip_fft_f    *plan,
                    const vsip_vview_f  *x,
                    const vsip_cvview_f *y);
```

Parameters

- plan, structure, input.
- x, real vector, input.
- y, complex vector, output.

Return Value

- none.

Description

Computes a real-to-complex out-of-place (forward) Fast Fourier Transform (FFT) of the real vector x, and stores the results in the complex vector y.

$y[k] := scale \cdot \sum_{j=0}^{N-1} x[j] \cdot (W_N)^{kj}$ where $W_N = \exp(-2\pi \mathrm{i}/N)$.

Restrictions

Only unit stride views are supported. The length, N, must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a real-to-complex out-of-place FFT object.
3. The input must be a real vector of even length N.
4. The output must be a complex vector of length $N/2 + 1$.
5. The input and output vectors must not overlap.

6. The input and output vectors must be unit stride.

Notes

The mathematical definition of the Fourier transform takes a sequence of N complex values and transforms it to another sequence of N complex values. A complex-to-complex FFT routine will take N complex inputs, and produce N complex outputs.

The purpose of a separate real-to-complex FFT routine is efficiency. Since the input data are real, you can make use of this fact to save almost half of the computational work. The theory of Fourier transforms tells us that for real input data, you have to compute only the first $\lfloor N/2 \rfloor + 1$ complex output values because the remaining values can be computed from the first half by the simple formula: $y_k = y^*_{N-k}$ for $\lfloor N/2 \rfloor < k < N$.

For real input data, the first output value, $y[0]$, will always be a real number (the imaginary part will be zero). The first output value is sometimes called the DC component of the FFT and corresponds to zero frequency. Since we restrict N to be an even number, $y[N/2]$ will also be real and thus, have zero imaginary part. The last value is called the folding frequency and is equal to one half the sample rate of the input data.

Thus, in the real-to-complex routine, x is a real array of even length N and y is a complex array of length $N/2 + 1$.

vsip_ccfftmip_create_f

Create a 1D multiple FFT object.

Prototype

```
vsip_fftm_f * vsip_ccfftmip_create_f(
                    const vsip_index    rows,
                    const vsip_index    cols,
                    const vsip_scalar_f scale,
                    const vsip_fft_dir  dir,
                    const vsip_major    major,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `rows`, vector-index scalar, input. Length of column FFT or number of row FFT's (M).
- `cols`, vector-index scalar, input. Length of row FFT or number of column FFT's (N).
- `scale`, real scalar, input. Typical scale factors are 1, $1/M$, $1/N$, $1/\sqrt{M}$ and $1/\sqrt{N}$.
- `dir`, enumerated type, input.

`VSIP_FFT_FWD`	forward
`VSIP_FFT_INV`	reverse (or inverse)

- `major`, enumerated type, input.

`VSIP_ROW`	apply operation to the rows
`VSIP_COL`	apply operation to the columns

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D multiple FFT object holding the information on the type of FFT to be computed: complex-to-complex in-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

A series of 1D real vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors.

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `rows` and `cols` must be positive.
2. `dir` must be valid.
3. `major` must be valid.
4. `hint` must be valid.

Notes

Performing a 1D FFT on the data by rows and then by columns (or vice versa) is equivalent to performing a 2D FFT on the matrix. This would require two multiple FFT objects, one by rows and one by columns.

FFT operations are supported for all values of N and M.

vsip_ccfftmop_create_f

Create a 1D multiple FFT object.

Prototype

```
vsip_fftm_f * vsip_ccfftmop_create_f(
                    const vsip_index    rows,
                    const vsip_index    cols,
                    const vsip_scalar_f scale,
                    const vsip_fft_dir  dir,
                    const vsip_major    major,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `rows`, vector-index scalar, input. Length of column FFT or number of row FFT's (M).
- `cols`, vector-index scalar, input. Length of row FFT or number of column FFT's (N).
- `scale`, real scalar, input. Typical scale factors are 1, $1/M$, $1/N$, $1/\sqrt{M}$ and $1/\sqrt{N}$.
- `dir`, enumerated type, input.

`VSIP_FFT_FWD`	forward
`VSIP_FFT_INV`	reverse (or inverse)

- `major`, enumerated type, input.

`VSIP_ROW`	apply operation to the rows
`VSIP_COL`	apply operation to the columns

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D multiple FFT object holding the information on the type of FFT to be computed: complex-to-complex out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

A series of 1D real vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors.

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `rows` and `cols` must be positive.
2. `dir` must be valid.
3. `major` must be valid.
4. `hint` must be valid.

Notes

Performing a 1D FFT on the data by rows and then by columns (or vice versa) is equivalent to performing a 2D FFT on the matrix. This would require two multiple FFT objects, one by rows and one by columns.

FFT operations are supported for all values of N and M.

vsip_crfftmop_create_f

Create a 1D multiple FFT object.

Prototype

```
vsip_fftm_f * vsip_crfftmop_create_f(
                    const vsip_index    rows,
                    const vsip_index    cols,
                    const vsip_scalar_f scale,
                    const vsip_major    major,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `rows`, vector-index scalar, input. Length of column FFT or number of row FFT's (M).
- `cols`, vector-index scalar, input. Length of row FFT or number of column FFT's (N).
- `scale`, real scalar, input. Typical scale factors are 1, $1/M$, $1/N$, $1/\sqrt{M}$ and $1/\sqrt{N}$.
- `major`, enumerated type, input.

`VSIP_ROW`	apply operation to the rows
`VSIP_COL`	apply operation to the columns

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D multiple FFT object holding the information on the type of FFT to be computed: (reverse) complex-to-real out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

A series of 1D real vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors.

`NULL` is returned if the create fails.

Restrictions

In the FFT direction, the length must be even and the stride must be 1.

Errors

The arguments must conform to the following:

1. `rows` and `cols` must be positive.
2. `major` must be valid.
3. `hint` must be valid.

Notes

Performing a 1D FFT on the data by rows and then by columns (or vice versa) is equivalent to performing a 2D FFT on the matrix. This would require two multiple FFT objects, one by rows and one by columns.

FFT operations are supported for all values of N and M.

vsip_rcfftmop_create_f

Create a 1D multiple FFT object.

Prototype

```
vsip_fftm_f * vsip_rcfftmop_create_f(
                    const vsip_index    rows,
                    const vsip_index    cols,
                    const vsip_scalar_f scale,
                    const vsip_major    major,
                    const vsip_length   ntimes,
                    const vsip_alg_hint hint);
```

Parameters

- `rows`, vector-index scalar, input. Length of column FFT or number of row FFT's (M).
- `cols`, vector-index scalar, input. Length of row FFT or number of column FFT's (N).
- `scale`, real scalar, input. Typical scale factors are 1, $1/M$, $1/N$, $1/\sqrt{M}$ and $1/\sqrt{N}$.
- `major`, enumerated type, input.

`VSIP_ROW`	apply operation to the rows
`VSIP_COL`	apply operation to the columns

- `ntimes`, integer scalar, input. An estimate of how many times the FFT object will be used. Zero is treated as 'many'.
- `hint`, enumerated type, input.

`VSIP_ALG_SPACE`	minimise memory usage
`VSIP_ALG_TIME`	minimise execution time
`VSIP_ALG_NOISE`	maximise numerical accuracy

Return Value

- structure.

Description

Creates a 1D multiple FFT object holding the information on the type of FFT to be computed: (forward) real-to-complex out-of-place. The 1D FFT object is used to compute a Fast Fourier Transform (FFT) of a vector x, and store the results in a vector y.

A series of 1D real vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors.

`NULL` is returned if the create fails.

Restrictions

In the FFT direction, the length must be even and the stride must be 1.

Errors

The arguments must conform to the following:

1. `rows` and `cols` must be positive.
2. `major` must be valid.
3. `hint` must be valid.

Notes

Performing a 1D FFT on the data by rows and then by columns (or vice versa) is equivalent to performing a 2D FFT on the matrix. This would require two multiple FFT objects, one by rows and one by columns.

FFT operations are supported for all values of N and M.

vsip_fftm_destroy_f

Destroy an FFT object.

Prototype

```
int vsip_fftm_destroy_f(
                    vsip_fftm_f  *plan);
```

Parameters

- `plan`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) an FFT object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The input object must conform to the following:

1. The FFT object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_fftm_getattr_f

Return the attributes of an FFT object.

Prototype

```
void vsip_fftm_getattr_f(
                    const vsip_fftm_f  *plan,
                    vsip_fftm_attr_f   *attr);
```

Parameters

- plan, structure, input.
- attr, pointer to structure, output.
  The attribute structure contains the following information:

vsip_scalar_mi	input	numbers of rows and columns in input matrix
vsip_scalar_mi	output	numbers of rows and columns in output matrix
vsip_fft_place	place	in-place or out-of-place
vsip_scalar_f	scale	scale factor
vsip_fft_dir	dir	forward or reverse
vsip_major	major	by row or column

Return Value

- none.

Description

Returns the attributes of an FFT object.

Restrictions

Errors

The arguments must conform to the following:

1. The FFT object plan must be valid.
2. The attribute pointer attr must not be NULL.

Notes

There is no attribute that explicitly indicates complex-to-complex, real-to-complex, or complex-to-real FFTs. This may be inferred by examining the input and output sizes.

vsip_ccfftmip_f

Apply a multiple complex-to-complex Fast Fourier Transform (FFT).

Prototype

```
void vsip_ccfftmip_f(
                    const vsip_fftm_f    *plan,
                    const vsip_cmview_f  *XY);
```

Parameters

- `plan`, structure, input.
- `XY`, complex matrix, modified in place.

Return Value

- none.

Description

Computes multiple complex-to-complex in-place Fast Fourier Transforms (FFTs) of the complex vectors in matrix X, and stores the results in the complex matrix Y.

A series of 1D complex vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors. The major direction (row or column) is specified in the creation of the FFT object.

See `vsip_ccfftip_split_f` for more details.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex in-place multiple FFT object.
3. The input must be a complex matrix of size M by N, conformant to the FFT object.

Notes

vsip_ccfftmop_f

Apply a multiple complex-to-complex Fast Fourier Transform (FFT).

Prototype

```
void vsip_ccfftmop_f(
                    const vsip_fftm_f    *plan,
                    const vsip_cmview_f  *X,
                    const vsip_cmview_f  *Y);
```

Parameters

- `plan`, structure, input.
- `X`, complex matrix, input.
- `Y`, complex matrix, output.

Return Value

- none.

Description

Computes multiple complex-to-complex out-of-place Fast Fourier Transforms (FFTs) of the complex vectors in matrix X, and stores the results in the complex matrix Y.

A series of 1D complex vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors. The major direction (row or column) is specified in the creation of the FFT object.

See `vsip_ccfftop_split_f` for more details.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex out-of-place multiple FFT object.
3. The input and output must be complex matrices of size M by N, conformant to the FFT object.
4. The input and output matrices must not overlap.

Notes

vsip_crfftmop_f

Apply a multiple complex-to-real Fast Fourier Transform (FFT).

Prototype

```
void vsip_crfftmop_f(
                    const vsip_fftm_f    *plan,
                    const vsip_cmview_f  *X,
                    const vsip_mview_f   *Y);
```

Parameters

- plan, structure, input.
- X, complex matrix, input.
- Y, real matrix, output.

Return Value

- none.

Description

Computes a complex-to-real out-of-place reverse Fast Fourier Transform (FFT) of the complex matrix X, and stores the results in the real matrix Y.

A series of 1D complex vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors (which must have unit stride). The major direction (row or column) is specified in the creation of the FFT object.

See vsip_crfftop_f for more details.

Restrictions

Only unit stride along the specified row or column FFT direction is supported. The output length of the individual FFTs must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.

2. The FFT object must be a complex-to-real out-of-place multiple FFT object.
3. The input must be a complex matrix of size:
   - By rows: M by $N/2+1$, N even.
   - By columns: $M/2+1$ by N, M even.

   M by N are obtained from the FFT object.
4. The output must be a real matrix of size M by N, conformant to the FFT object.
5. The input and output matrices must not overlap.
6. The input and output matrices must be unit-stride in the transform direction.

Notes

vsip_rcfftmop_f

Apply a multiple real-to-complex out of place Fast Fourier Transform (FFT).

Prototype

```
void vsip_rcfftmop_f(
                    const vsip_fftm_f    *plan,
                    const vsip_mview_f   *X,
                    const vsip_cmview_f  *Y);
```

Parameters

- plan, structure, input.
- X, real matrix, input.
- Y, complex matrix, output.

Return Value

- none.

Description

Computes a real-to-complex out-of-place (forward) Fast Fourier Transform (FFT) of the real matrix X, and stores the results in the complex matrix Y.

A series of 1D real vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors (which must have unit stride). The major direction (row or column) is specified in the creation of the FFT object.

See vsip_rcfftop_f for more details.

Restrictions

Only unit stride along the specified row or column FFT direction is supported. The input length of the individual FFTs must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.

2. The FFT object must be a real-to-complex out-of-place multiple FFT object.
3. The output must be a complex matrix of size:
   - By rows: M by $N/2+1$, N even.
   - By columns: $M/2+1$ by N, M even.

   M by N are obtained from the FFT object.
4. The input must be a real matrix of size M by N, conformant to the FFT object.
5. The input and output matrices must not overlap.
6. The input and output matrices must be unit-stride in the transform direction.

Notes

7.2 Convolution/Correlation Functions

- vsip_conv1d_create_f
- vsip_conv1d_destroy_f
- vsip_conv1d_getattr_f
- vsip_convolve1d_f
- vsip_Dcorr1d_create_P
- vsip_Dcorr1d_destroy_P
- vsip_Dcorr1d_getattr_P
- vsip_Dcorrelate1d_P

vsip_conv1d_create_f

Create a decimated 1D convolution filter object.

Prototype

```
vsip_conv1d_f * vsip_conv1d_create_f(
                    const vsip_vview_f         *kernel,
                    const vsip_symmetry         symm,
                    const vsip_length           N,
                    const vsip_length           D,
                    const vsip_support_region   support,
                    const vsip_length           ntimes,
                    const vsip_alg_hint         hint);
```

Parameters

- kernel, real vector, input. Vector of non-redundant filter coefficients. There are M in the non-symmetric case and $\lceil M/2 \rceil$ for symmetric filters.
- symm, enumerated type, input.

VSIP_NONSYM	non-symmetric
VSIP_SYM_EVEN_LEN_ODD	(even) symmetric, odd length
VSIP_SYM_EVEN_LEN_EVEN	(even) symmetric, even length

- N, integer scalar, input. Length of data vector.
- D, integer scalar, input. Decimation factor.
- support, enumerated type, input.

VSIP_SUPPORT_FULL	maximum region
VSIP_SUPPORT_SAME	input and output same size
VSIP_SUPPORT_MIN	region without zero extending the kernel

- ntimes, integer scalar, input. An estimate of how many times the filter will be used. Zero is treated as 'many'.
- hint, enumerated type, input.

VSIP_ALG_SPACE	minimise memory usage
VSIP_ALG_TIME	minimise execution time
VSIP_ALG_NOISE	maximise numerical accuracy

Return Value

- structure.

Description

Creates a decimated convolution filter object and returns a pointer to the object. The user specifies the kernel (filter order, symmetry, and filter coefficients), the region of support, and the integral output decimation factor.

If the create fails, `NULL` is returned.

A 1D convolution object is used to compute the convolution of a real filter (kernel) vector h of length M with a real data vector x of length N and output decimation factor D to produce an output vector y.

Full:
length $= \lfloor (N + M - 2)/D \rfloor + 1$

$$y[k] := \sum_{j=0}^{M-1} h[j]\,\langle x[kD - j]\rangle \qquad \text{for } 0 \leqslant k \leqslant \lfloor (N + M - 2)/D \rfloor.$$

Same size:
length $= \lfloor (N - 1)/D \rfloor + 1$

$$y[k] := \sum_{j=0}^{M-1} h[j]\langle x[kD + \lfloor M/2 \rfloor - j]\rangle \qquad \text{for } 0 \leqslant k \leqslant \lfloor (N - 1)/D \rfloor.$$

Minimum (not zero padded):
length $= \lfloor (N - 1)/D \rfloor - \lfloor (M - 1)/D \rfloor + 1$

$$y[k] := \sum_{j=0}^{M-1} h[j]\langle x[kD + (M - 1) - j]\rangle \qquad \text{for } 0 \leqslant k \leqslant \lfloor (N - 1)/D \rfloor - \lfloor (M - 1)/D \rfloor.$$

Where $\langle x[j]\rangle = \begin{cases} x[j] & : \quad 0 \leqslant j < N \\ 0 & : \quad \text{otherwise.} \end{cases}$

If the kernel is symmetric the redundant values are not specified.

Restrictions

The filter length must be less than or equal to the data length, $M \leqslant N$.

Errors

The arguments must conform to the following:

1. `kernel` must be a pointer to a valid vector view object.
2. `symm` must be valid.
3. `N` must be greater than or equal to the filter length M.
4. `D` $\geqslant 1$.

5. `support` must be valid.
6. `hint` must be valid.

Notes

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments. The decimation factor, D, is normally one for non-lowpass filters.

vsip_conv1d_destroy_f

Destroy a 1D convolution object.

Prototype

```
int vsip_conv1d_destroy_f(
                    vsip_conv1d_f  *plan);
```

Parameters

- `plan`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) a 1D convolution object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D convolution object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_conv1d_getattr_f

Returns the attributes for a 1D convolution object.

Prototype

```
void vsip_conv1d_getattr_f(
                    const vsip_conv1d_f  *plan,
                    vsip_conv1d_attr_f   *attr);
```

Parameters

- plan, structure, input.
- attr, pointer to structure, output.
  The attribute structure contains the following information:

vsip_scalar_vi	kernel_len	kernel length
vsip_symmetry	symm	kernel symmetry
vsip_scalar_vi	data_len	data input length
vsip_support_region	support	output region of support
vsip_scalar_vi	out_len	output length
vsip_length	decimation	output decimation factor

Return Value

- none.

Description

Returns the attributes for a 1D convolution object.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D convolution object plan must be valid.
2. The attribute pointer attr must not be NULL.

Notes

The length of the kernel is also known as the filter order.

vsip_convolve1d_f

Compute a decimated real one-dimensional (1D) convolution of two vectors.

Prototype

```
void vsip_convolve1d_f(
                    const vsip_conv1d_f  *plan,
                    const vsip_vview_f   *x,
                    const vsip_vview_f   *y);
```

Parameters

- plan, structure, input.
- x, real vector, input.
- y, real vector, output.

Return Value

- none.

Description

Uses a 1D convolution object to compute the convolution of a real filter (kernel) vector h of length M with a real data vector x of length N and output decimation factor D to produce an output vector y.

See vsip_conv1d_create_f for details.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D convolution object plan must be valid.
2. The input vector x must be of length N (conformant with the 1D convolution object).
3. The output vector y must be of length (conformant with the 1D convolution object):
   - Full: $\lfloor (N + M - 2)/D \rfloor + 1$
   - Same: $\lfloor (N - 1)/D \rfloor + 1$

- Minimum: $\lfloor (N-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$.

4. The input `x`, and the output `y`, must not overlap.

Notes

The decimation factor, D, is normally one for non-lowpass filters.

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments.

vsip_Dcorr1d_create_P

Create a 1D correlation object.

Prototype

```
vsip_Dcorr1d_P * vsip_Dcorr1d_create_P(
                     const vsip_length         M,
                     const vsip_length         N,
                     const vsip_support_region support,
                     const vsip_length         ntimes,
                     const vsip_alg_hint       hint);
```

The following instances are supported:

vsip_corr1d_create_f

vsip_ccorr1d_create_f

Parameters

- M, integer scalar, input. Length of input reference vector.
- N, integer scalar, input. Length of input data vector.
- support, enumerated type, input.

VSIP_SUPPORT_FULL	maximum region
VSIP_SUPPORT_SAME	input and output same size
VSIP_SUPPORT_MIN	region without zero extending the kernel

  Output region of support (indicates which output points are computed).
- ntimes, integer scalar, input. An estimate of how many times the filter will be used. Zero is treated as 'many'.
- hint, enumerated type, input.

VSIP_ALG_SPACE	minimise memory usage
VSIP_ALG_TIME	minimise execution time
VSIP_ALG_NOISE	maximise numerical accuracy

Return Value

- structure.

Description

Creates a (cross-) correlation object and returns a pointer to the object. The user specifies the lengths of the reference vector r and the data vector x.

If the create fails, `NULL` is returned.

A 1D correlation object is used to compute the (cross-) correlation of a reference vector r of length M with a data vector x of length N to produce an output vector y.

Full:
length $= N + M - 1$

$$\widehat{y}[k] := \sum_{j=0}^{M-1} r[j]\,\langle x[k+j-M+1]^*\rangle \qquad \text{for } 0 \leqslant k \leqslant N+M-2$$

$$y[k] := \widehat{y}[k] * \begin{cases} 1/(k+1) & : \quad 0 \leqslant k < M-1 \\ 1/M & : \quad M-1 \leqslant k < N \\ 1/(N+M-1-k) & : \quad N \leqslant k < N+M-1. \end{cases}$$

Same size:
length $= N$

$$\widehat{y}[k] := \sum_{j=0}^{M-1} r[j]\langle x[k+j-\lfloor M/2\rfloor]^*\rangle \qquad \text{for } 0 \leqslant k \leqslant N-1$$

$$y[k] := \widehat{y}[k] * \begin{cases} 1/(k+\lceil M/2\rceil) & : \quad 0 \leqslant k < \lfloor M/2\rfloor \\ 1/M & : \quad \lfloor M/2\rfloor \leqslant k < N-\lfloor M/2\rfloor \\ 1/(N+\lfloor M/2\rfloor-k) & : \quad N-\lfloor M/2\rfloor \leqslant k < N. \end{cases}$$

Minimum (not zero padded):
length $= N - M + 1$

$$\widehat{y}[k] := \sum_{j=0}^{M-1} r[j]\langle x[k+j]^*\rangle \qquad \text{for } 0 \leqslant k \leqslant N-M$$

$$y[k] := \widehat{y}[k]/M.$$

Where $\langle x[j]\rangle = \begin{cases} x[j] & : \quad 0 \leqslant j < N \\ 0 & : \quad \text{otherwise.} \end{cases}$

The values $\widehat{y}[k]$ are the biased correlation estimates while the $y[k]$ are unbiased estimates. (The unbiased estimates are scaled by the number or terms in the summation for each lag where $\langle x[j]\rangle$ is not defined to be zero.)

Restrictions

The reference length must be less than or equal to the data length, $M \leqslant N$.

Errors

The arguments must conform to the following:

1. $1 \leqslant$ `M` $\leqslant$ `N`.
2. `support` must be valid.

3. `hint` must be valid.

Notes

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments: specify the FIR kernel as the reverse-indexed clone of the reference data.

vsip_Dcorr1d_destroy_P

Destroy a 1D correlation object.

Prototype

```
int vsip_Dcorr1d_destroy_P(
                    vsip_Dcorr1d_P *plan);
```

The following instances are supported:

```
vsip_corr1d_destroy_f
vsip_ccorr1d_destroy_f
```

Parameters

- `plan`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) a 1D correlation object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D correlation object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_Dcorr1d_getattr_P

Return the attributes for a 1D correlation object.

Prototype

```
void vsip_Dcorr1d_getattr_P(
                    const vsip_Dcorr1d_P  *plan,
                    vsip_Dcorr1d_attr_P   *attr);
```

The following instances are supported:

```
vsip_corr1d_getattr_f
vsip_ccorr1d_getattr_f
```

Parameters

- plan, structure, input.
- attr, pointer to structure, output.
  The attribute structure contains the following information:

vsip_scalar_vi	ref_len	reference length
vsip_scalar_vi	data_len	data input length
vsip_support_region	support	output region of support
vsip_scalar_vi	lag_len	output (lags) length

Return Value

- none.

Description

Returns the attributes for a 1D correlation object.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D correlation object plan must be valid.
2. The attribute pointer attr must not be NULL.

Notes

vsip_Dcorrelate1d_P

Compute a real one-dimensional (1D) correlation of two vectors.

Prototype

```
void vsip_Dcorrelate1d_P(
                    const vsip_Dcorr1d_P *plan,
                    const vsip_bias       bias,
                    const vsip_Dvview_P  *ref,
                    const vsip_Dvview_P  *x,
                    const vsip_Dvview_P  *y);
```

The following instances are supported:

`vsip_correlate1d_f`

`vsip_ccorrelate1d_f`

Parameters

- `plan`, structure, input.
- `bias`, enumerated type, input.

`VSIP_BIASED`	biased
`VSIP_UNBIASED`	unbiased

  Return biased or unbiased estimates.
- `ref`, real or complex vector, input.
- `x`, real or complex vector, input.
- `y`, real or complex vector, output.

Return Value

- none.

Description

Uses a 1D correlation object to compute the (cross-) correlation of a reference vector r of length M with a data vector x of length N to produce an output vector y.

See `vsip_Dcorr1d_create_P` for details.

Restrictions

The reference length must be less than or equal to the data length, $M \leqslant N$.

Errors

The arguments must conform to the following:

1. The 1D correlation object `plan` must be valid.
2. `bias` must be valid.
3. The reference input vector `ref` must be of length M (conformant with the 1D correlation object).
4. The data input vector `x` must be of length N (conformant with the 1D correlation object).
5. The output vector `y` must be of length (conformant with the 1D correlation object):
   - Full: $N + M - 1$
   - Same: N
   - Minimum: $N - M + 1$.
6. The output `y` cannot overlap either of the input vector, `ref` or `x`.

Notes

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments: specify the FIR kernel as the reverse-indexed clone of the reference data.

7.3 Window Functions

- vsip_vcreate_blackman_f
- vsip_vcreate_cheby_f
- vsip_vcreate_hanning_f
- vsip_vcreate_kaiser_f

vsip_vcreate_blackman_f

Create a vector with Blackman window weights.

Prototype

```
vsip_vview_f * vsip_vcreate_blackman_f(
                    const vsip_length      N,
                    const vsip_memory_hint hint);
```

Parameters

- N, integer scalar, input. Length of window.
- hint, enumerated type, input.

VSIP_MEM_NONE	no hint
VSIP_MEM_RDONLY	read-only
VSIP_MEM_CONST	constant
VSIP_MEM_SHARED	shared
VSIP_MEM_SHARED_RDONLY	shared and read-only
VSIP_MEM_SHARED_CONST	shared and constant

Return Value

- real vector.

Description

Creates a vector initialised with a Blackman window of length N.

$$X[k] := 0.42 - 0.5\cos\left(\frac{2\pi k}{N-1}\right) + 0.8\cos\left(\frac{4\pi k}{N-1}\right).$$

NULL is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. N > 1.
2. hint must be valid.

Notes

vsip_vcreate_cheby_f

Create a vector with Dolph-Chebyshev window weights.

Prototype

```
vsip_vview_f * vsip_vcreate_cheby_f(
                     const vsip_length      N,
                     const vsip_scalar_f    ripple,
                     const vsip_memory_hint hint);
```

Parameters

- `N`, integer scalar, input. Length of window.
- `ripple`, real scalar, input. Side lobes are this number of decibels below the main lobe.
- `hint`, enumerated type, input.

`VSIP_MEM_NONE`	no hint
`VSIP_MEM_RDONLY`	read-only
`VSIP_MEM_CONST`	constant
`VSIP_MEM_SHARED`	shared
`VSIP_MEM_SHARED_RDONLY`	shared and read-only
`VSIP_MEM_SHARED_CONST`	shared and constant

Return Value

- real vector.

Description

Creates a vector initialised with a Dolph-Chebyshev window of length N.

$$\delta_p := 10^{-ripple/20}$$

$$\tau_p := \frac{1+\delta_p}{\delta_p}$$

$$\delta_f := \frac{1}{\pi}\ \cos^{-1}\left[\frac{1}{\cosh\left(\frac{\cosh^{-1}(\tau_p)}{N-1}\right)}\right]$$

$$x[0] := \frac{3-\cos(2\pi\delta_f)}{1+\cos(2\pi\delta_f)}$$

$$x[k] := \frac{x[0]+1}{2}\ \left(\cos\left(\frac{2\pi k}{N}\right)\right) + \frac{x[0]-1}{2}$$

N odd:

$$W[k] := \begin{cases} \delta_p \cosh\left(\frac{N-1}{2} \cosh^{-1}(x[k])\right) & : \quad |x[k]| > 1 \\ \delta_p \cos\left(\frac{N-1}{2} \cos^{-1}(x[k])\right) & : \quad |x[k]| \leqslant 1 \end{cases}$$

N even:

$$W[k] := \begin{cases} \delta_p \cosh\left(\frac{N-1}{2} \cosh^{-1}(x[k])\right) \exp(\pi \mathrm{i} k/N) & : \quad |x[k]| > 1, \quad 0 \leqslant k \leqslant \lfloor N/2 \rfloor \\ -\delta_p \cosh\left(\frac{N-1}{2} \cosh^{-1}(x[k])\right) \exp(\pi \mathrm{i} k/N) & : \quad |x[k]| > 1, \quad \lfloor N/2 \rfloor < k < N \\ \delta_p \cos\left(\frac{N-1}{2} \cos^{-1}(x[k])\right) \exp(\pi \mathrm{i} k/N) & : \quad |x[k]| \leqslant 1, \quad 0 \leqslant k \leqslant \lfloor N/2 \rfloor \\ -\delta_p \cos\left(\frac{N-1}{2} \cos^{-1}(x[k])\right) \exp(\pi \mathrm{i} k/N) & : \quad |x[k]| \leqslant 1, \quad \lfloor N/2 \rfloor < k < N. \end{cases}$$

FFT the W's:

$$w[k] := \sum_{j=0}^{N-1} W[k] \exp(2\pi \mathrm{i} jk/N).$$

Populate the window with the frequency swap of the w's:

$$X[k] := \begin{cases} \text{real}\left(\frac{w[k+\lfloor N/2 \rfloor]}{w[0]}\right) & : \quad 0 \leqslant k < \lfloor N/2 \rfloor \\ \text{real}\left(\frac{w[k-\lfloor N/2 \rfloor]}{w[0]}\right) & : \quad \lfloor N/2 \rfloor \leqslant k < N. \end{cases}$$

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `N` > 0.
2. `hint` must be a valid.

Notes

vsip_vcreate_hanning_f

Create a vector with Hanning window weights.

Prototype

```
vsip_vview_f * vsip_vcreate_hanning_f(
                    const vsip_length      N,
                    const vsip_memory_hint hint);
```

Parameters

- N, integer scalar, input. Length of window.
- hint, enumerated type, input.

VSIP_MEM_NONE	no hint
VSIP_MEM_RDONLY	read-only
VSIP_MEM_CONST	constant
VSIP_MEM_SHARED	shared
VSIP_MEM_SHARED_RDONLY	shared and read-only
VSIP_MEM_SHARED_CONST	shared and constant

Return Value

- real vector.

Description

Creates a vector initialised with a Hanning window of length N.

$$X[k] := 0.5\left(1 - \cos\left(\frac{2\pi(k+1)}{N+1}\right)\right).$$

NULL is returned if the create fails.

Restrictions

Restrictions

Errors

The arguments must conform to the following:

1. N > 1.
2. hint must be valid.

Notes

There are two different widely used definitions of the Hanning window. The other is

$$X[k] := 0.5\left(1 - \cos\left(\frac{2\pi k)}{N-1}\right)\right).$$

This form has a weight of zero for both end points of the window; we use the form that does not have zero end points.

If you want the window to be periodic of length N, you must generate a Hanning window of length $N-1$, copy it to a vector of length N, and set the last point to 0.0.

vsip_vcreate_kaiser_f

Create a vector with Kaiser window weights.

Prototype

```
vsip_vview_f * vsip_vcreate_kaiser_f(
                    const vsip_length      N,
                    const vsip_scalar_f    beta,
                    const vsip_memory_hint hint);
```

Parameters

- `N`, integer scalar, input. Length of window.
- `beta`, real scalar, input. Transition width.
- `hint`, enumerated type, input.

`VSIP_MEM_NONE`	no hint
`VSIP_MEM_RDONLY`	read-only
`VSIP_MEM_CONST`	constant
`VSIP_MEM_SHARED`	shared
`VSIP_MEM_SHARED_RDONLY`	shared and read-only
`VSIP_MEM_SHARED_CONST`	shared and constant

Return Value

- real vector.

Description

Creates a vector initialised with a Kaiser window of length N.

$$X[k] := \frac{I_0\left[\beta\sqrt{1-\left(\frac{2k-N+1}{N-1}\right)^2}\right]}{I_0[\beta]} \text{ where } I_0[x] = \sum_{p=0}^{\infty}\left(\frac{x^p}{2^p\,p!}\right)^2.$$

Increasing β widens the main lobe (transition width) and reduces the side lobes.

`NULL` is returned if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. `N` > 1.
2. `hint` must be valid.

Notes

7.4 Filter Functions

- vsip_Dfir_create_P
- vsip_Dfir_destroy_P
- vsip_Dfirflt_P
- vsip_Dfir_getattr_P

vsip_Dfir_create_P

Create a decimated FIR filter object.

Prototype

```
vsip_Dfir_P * vsip_Dfir_create_P(
                    const vsip_Dvview_P   *kernel,
                    const vsip_symmetry    symm,
                    const vsip_length      N,
                    const vsip_length      D,
                    const vsip_obj_state   state,
                    const vsip_length      ntimes,
                    const vsip_alg_hint    hint);
```

The following instances are supported:

vsip_fir_create_f

vsip_cfir_create_f

Parameters

- kernel, real or complex vector, input. Vector of non-redundant filter coefficients. There are $M+1$ in the non-symmetric case and $\lceil (M+1)/2 \rceil$ for symmetric filters.
- symm, enumerated type, input.

VSIP_NONSYM	non-symmetric
VSIP_SYM_EVEN_LEN_ODD	(even) symmetric, odd length
VSIP_SYM_EVEN_LEN_EVEN	(even) symmetric, even length

- N, integer scalar, input. Length of data vector.
- D, integer scalar, input. Decimation factor.
- state, enumerated type, input.

VSIP_STATE_NO_SAVE	do not save state — single call filter
VSIP_STATE_SAVE	save state for continuous filter

- ntimes, integer scalar, input. An estimate of how many times the filter will be used. Zero is treated as 'many'.
- hint, enumerated type, input.

VSIP_ALG_SPACE	minimise memory usage
VSIP_ALG_TIME	minimise execution time
VSIP_ALG_NOISE	maximise numerical accuracy

Return Value

- structure.

Description

Creates a decimated FIR filter object and returns a pointer to the object. The user specifies the kernel (filter coefficients and filter order), the integral output decimation factor, D, the length of the input segments (vectors) that will be filtered, and whether to save state information for continuous filtering.

If the create fails, `NULL` is returned.

If requested, the FIR filter object encapsulates the filter's state information. The state is initialised to zero. The filter state allows long data streams to be processed in segments by successive calls to `vsip_Dfirflt_P`.

The FIR filter object is used to compute:

$$y[k] := \sum_{j=0}^{M} h[j]\ \widehat{x}[p+kD-j] \text{ where } \widehat{x}[j] = \begin{cases} s[j] & : \quad j < 0 \\ x[j] & : \quad j \geqslant 0 \end{cases}, \text{ and } 0 \leqslant k < \lceil (N-p)/D \rceil.$$

The vector s and integer p are private internal state information. When the FIR filter object is created they are initialised to zero, and they will remain so if the SAVE option is not specified. Otherwise
$s[j] := x[N+j]$ for $-M \leqslant j < 0$ and
$p := D - 1 - [(N-1-p) \bmod D]$.

Given a filter kernel of order M with coefficient vector h, segment length N, and decimation factor D, the decimated output y is of length $(N-p)/D$.

Restrictions

The decimation factor must be less than or equal to the filter length.

Errors

The arguments must conform to the following:

1. kernel must be a pointer to a valid vector.
2. `symm` must be valid.
3. $N \geqslant M$.
4. $1 \leqslant D \leqslant M$.
5. `state` must be valid.
6. `hint` must be valid.

Notes

For non-lowpass filters, set $D = 1$.

It is important that the kernel vector be only as long as necessary (see above) — the symmetric values of the filter between the kernel's centre and its last value are not to be included in the kernel.

It is safe to destroy the kernel after creating the FIR filter object.

The filter will be evaluated directly unless `hint` is `VSIP_ALG_TIME`, in which case convolution in the frequency domain (a method based on FFTs) will be used if it is quicker.

vsip_Dfir_destroy_P

Destroy a FIR filter object.

Prototype

```
int vsip_Dfir_destroy_P(
                    vsip_Dfir_P  *plan);
```

The following instances are supported:

```
vsip_fir_destroy_f
vsip_cfir_destroy_f
```

Parameters

- `plan`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) a FIR filter object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The FIR filter object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_Dfirflt_P

FIR filter an input sequence and decimate the output.

Prototype

```
int vsip_Dfirflt_P(
                    vsip_Dfir_P          *plan,
                    const vsip_Dvview_P  *x,
                    const vsip_Dvview_P  *y);
```

The following instances are supported:

vsip_firflt_f

vsip_cfirflt_f

Parameters

- plan, structure, input.
- x, real or complex vector, input.
- y, real or complex vector, output.

Return Value

- number of values computed.

Description

Applies a FIR filter, specified by the FIR filter object, to an input segment x, and computes a decimated output segment y. Initial and final filter state is encapsulated in the FIR filter object. Long data streams can be processed in segments by successive calls to this function.

When N is a multiple of D, the length of y and the number of output samples is N/D.

When N is not a multiple of D, the length of y is $\lceil N/D \rceil$, although it may not be fully populated; there may be only $\lfloor N/D \rfloor$ values.

The return value is the number of output samples computed.

Restrictions

Filtering cannot be performed in place.

Errors

The arguments must conform to the following:

1. The FIR filter object must be valid.
2. The input vector `x` must be of length N (conformant with the FIR filter object).
3. The output vector `y` must be of length $\lceil N/D \rceil$ (conformant with the FIR filter object).
4. The input `x`, and the output `y`, must not overlap.

Notes

The filter object may be modified with the updated state.

vsip_Dfir_getattr_P

Return the attributes of a FIR filter object.

Prototype

```
void vsip_Dfir_getattr_P(
                    const vsip_Dfir_P *plan,
                    vsip_Dfir_attr_P  *attr);
```

The following instances are supported:

```
vsip_fir_getattr_f
vsip_cfir_getattr_f
```

Parameters

- `plan`, structure, input.
- `attr`, pointer to structure, output.
  The attribute structure contains the following information:

`vsip_scalar_vi`	`kernel_len`	kernel length
`vsip_symmetry`	`symm`	kernel symmetry
`vsip_scalar_vi`	`in_len`	filter input segment length
`vsip_scalar_vi`	`out_len`	filter output segment length
`vsip_length`	`decimation`	decimation factor
`vsip_obj_state`	`state`	save state information

Return Value

- none.

Description

Returns the attributes of a FIR filter object.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object `plan` must be valid.
2. The attribute pointer `attr` must not be `NULL`.

Notes

The filter coefficient values are not accessible attributes. For a symmetric kernel, the filter kernel length, $M + 1$, is not the length of the vector view, $\lceil (M + 1)/2 \rceil$.

7.5 Miscellaneous Signal Processing Functions

- vsip_vhisto_f

vsip_vhisto_f

Compute the histogram of a vector.

Prototype

```
void vsip_vhisto_f(
                    const vsip_vview_f   *A,
                    const vsip_scalar_f   min,
                    const vsip_scalar_f   max,
                    const vsip_hist_opt   opt,
                    const vsip_vview_f   *R);
```

Parameters

- A, real vector, length n, input.
- min, real scalar, input.
- max, real scalar, input.
- opt, enumerated type, input.

  VSIP_HIST_RESET reset histogram each time
  VSIP_HIST_ACCUM accumulate histogram
- R, real vector, output.

Return Value

- none.

Description

Computes the histogram of a vector. Suppose the number of bins in the output vector is P. The first and last elements of the output vector are used to accumulate values outside the range of interest. The bin size is determined from the remaining $P-2$ bins and the boundary values.

The output vector is initialised to zero if the RESET option is used, otherwise the histogram is accumulated on top of the current data in the output vector.

The bin b is assigned as follows: if $\mathtt{A}[j] < \mathtt{min}$ then $b := 0$ else if $\mathtt{A}[j] \geqslant \mathtt{max}$ then $b := P-1$ else $b := 1 + \lfloor (P-2)(\mathtt{A}[j] - \mathtt{min})/(\mathtt{max} - \mathtt{min}) \rfloor$.

Restrictions

Errors

The arguments must conform to the following:

1. All the vector objects must be valid and of positive length.
2. `min` < `max`.

Notes

The first and last bins collect all the values less than `min`, and greater or equal to `max`, respectively. If these outlier values are not desired, create and bind a view of length P, and create a derived view of the first view starting at index 1 and of length $P - 2$. Collect the histogram into the larger view. The histogram values without the outliers are available in the derived view.

Chapter 8. Linear Algebra

8.1 Matrix and Vector Operations

- vsip_cmherm_f
- vsip_cvjdot_f
- vsip_gemp_f
- vsip_cgemp_f
- vsip_gems_f
- vsip_cgems_f
- vsip_Dmprod_P
- vsip_cmprodh_f
- vsip_cmprodj_f
- vsip_Dmprodt_P
- vsip_Dmvprod_P
- vsip_Dmtrans_P
- vsip_Dvdot_P
- vsip_Dvmprod_P
- vsip_vouter_f
- vsip_cvouter_f

vsip_cmherm_f

Complex Hermitian (conjugate transpose) of a matrix.

Prototype

```
void vsip_cmherm_f(
                    const vsip_cmview_f  *A,
                    const vsip_cmview_f  *R);
```

Parameters

- A, complex matrix, size m by n, input.
- R, complex matrix, size n by m, output.

Return Value

- none.

Description

R := A^H.

Restrictions

If the matrix A is square, the transpose is in place if A and R resolve to the same object, otherwise A and R must be disjoint.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices A and R must be conformant.
3. If the matrix is not square, the input and output matrix views must not overlap. If the matrix is square, the input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_cvjdot_f

Compute the conjugate inner (dot) product of two complex vectors.

Prototype

```
vsip_cscalar_f vsip_cvjdot_f(
                    const vsip_cvview_f  *A,
                    const vsip_cvview_f  *B);
```

Parameters

- A, complex vector, length n, input.
- B, complex vector, length n, input.

Return Value

- complex scalar.

Description

return value := $A^T \cdot B^*$.

Restrictions

Overflow may occur.

Errors

The arguments must conform to the following:

1. Arguments for input must be the same size.
2. All view objects must be valid.

Notes

vsip_gemp_f

Calculate the general product of two matrices and accumulate.

Prototype

```
void vsip_gemp_f(
                    const vsip_scalar_f  alpha,
                    const vsip_mview_f  *A,
                    const vsip_mat_op    Aop,
                    const vsip_mview_f  *B,
                    const vsip_mat_op    Bop,
                    const vsip_scalar_f  beta,
                    const vsip_mview_f  *R);
```

Parameters

- alpha, real scalar, input.
- A, real matrix, size m by p, input. The size shows the dimensions after Aop has been applied.
- Aop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose

- B, real matrix, size p by n, input. The size shows the dimensions after Bop has been applied.
- Bop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose

- beta, real scalar, input.
- R, real matrix, size m by n, output.

Return Value

- none.

Description

R := alpha · Aop(A) · Bop(B) + beta · R
where Aop and Bop are matrix operations

(identity or transpose).

Restrictions

The result matrix view, may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The input and output matrix views must not overlap.
3. The matrices must be conformant.
4. `Aop` and `Bop` must be valid.

Notes

vsip_cgemp_f

Calculate the general product of two matrices and accumulate.

Prototype

```
void vsip_cgemp_f(
                    const vsip_cscalar_f  alpha,
                    const vsip_cmview_f  *A,
                    const vsip_mat_op     Aop,
                    const vsip_cmview_f  *B,
                    const vsip_mat_op     Bop,
                    const vsip_cscalar_f  beta,
                    const vsip_cmview_f  *R);
```

Parameters

- alpha, complex scalar, input.
- A, complex matrix, size m by p, input. The size shows the dimensions after Aop has been applied.
- Aop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose
VSIP_MAT_HERM	Hermitian (conjugate transpose)
VSIP_MAT_CONJ	conjugate

- B, complex matrix, size p by n, input. The size shows the dimensions after Bop has been applied.
- Bop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose
VSIP_MAT_HERM	Hermitian (conjugate transpose)
VSIP_MAT_CONJ	conjugate

- beta, complex scalar, input.
- R, complex matrix, size m by n, output.

Return Value

- none.

Description

R := alpha · Aop(A) · Bop(B) + beta · R
where Aop and Bop are matrix operations
(identity, transpose, conjugate or Hermitian).

Restrictions

The result matrix view, may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The input and output matrix views must not overlap.
3. The matrices must be conformant.
4. Aop and Bop must be valid.

Notes

vsip_gems_f

Calculate a general matrix sum.

Prototype

```
void vsip_gems_f(
                    const vsip_scalar_f  alpha,
                    const vsip_mview_f  *A,
                    const vsip_mat_op    Aop,
                    const vsip_scalar_f  beta,
                    const vsip_mview_f  *C);
```

Parameters

- alpha, real scalar, input.
- A, real matrix, size m by n, input. The size shows the dimensions after Aop has been applied.
- Aop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose

- beta, real scalar, input.
- C, real matrix, size m by n, output.

Return Value

- none.

Description

$\texttt{C} := \texttt{alpha} \cdot \texttt{Aop}(\texttt{A}) + \texttt{beta} \cdot \texttt{C}$
where Aop is a matrix operation (identity or transpose).

Restrictions

The result matrix view C may not overlap the input matrix view A.

Errors

The arguments must conform to the following:

1. All objects must be valid.

2. The input and output matrix views must not overlap.
3. The matrices must be conformant.
4. `Aop` must be valid.

Notes

vsip_cgems_f

Calculate a general matrix sum.

Prototype

```
void vsip_cgems_f(
                    const vsip_cscalar_f  alpha,
                    const vsip_cmview_f  *A,
                    const vsip_mat_op     Aop,
                    const vsip_cscalar_f  beta,
                    const vsip_cmview_f  *C);
```

Parameters

- alpha, complex scalar, input.
- A, complex matrix, size m by n, input. The size shows the dimensions after Aop has been applied.
- Aop, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose
VSIP_MAT_HERM	Hermitian (conjugate transpose)
VSIP_MAT_CONJ	conjugate

- beta, complex scalar, input.
- C, complex matrix, size m by n, output.

Return Value

- none.

Description

$\texttt{C} := \texttt{alpha} \cdot \texttt{Aop}(\texttt{A}) + \texttt{beta} \cdot \texttt{C}$
where Aop is a matrix operation (identity, transpose, conjugate or Hermitian).

Restrictions

The result matrix view C may not overlap the input matrix view A.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The input and output matrix views must not overlap.
3. The matrices must be conformant.
4. `Aop` must be valid.

Notes

vsip_Dmprod_P

Calculate the product of two matrices.

Prototype

```
void vsip_Dmprod_P(
                    const vsip_Dmview_P  *A,
                    const vsip_Dmview_P  *B,
                    const vsip_Dmview_P  *R);
```

The following instances are supported:

vsip_mprod_f

vsip_cmprod_f

Parameters

- A, real or complex matrix, size m by p, input.
- B, real or complex matrix, size p by n, input.
- R, real or complex matrix, size m by n, output.

Return Value

- none.

Description

R := A · B.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices must be conformant.
3. The input and output matrix views must not overlap.

Notes

vsip_cmprodh_f

Calculate the product a complex matrix and the Hermitian of a complex matrix.

Prototype

```
void vsip_cmprodh_f(
                    const vsip_cmview_f  *A,
                    const vsip_cmview_f  *B,
                    const vsip_cmview_f  *R);
```

Parameters

- A, complex matrix, size m by p, input.
- B, complex matrix, size n by p, input.
- R, complex matrix, size m by n, output.

Return Value

- none.

Description

$R := A \cdot B^{H}$.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices A, B and R must be conformant.
3. The input and output matrix views must not overlap.

Notes

vsip_cmprodj_f

Calculate the product a complex matrix and the conjugate of a complex matrix.

Prototype

```
void vsip_cmprodj_f(
                    const vsip_cmview_f  *A,
                    const vsip_cmview_f  *B,
                    const vsip_cmview_f  *R);
```

Parameters

- A, complex matrix, size m by p, input.
- B, complex matrix, size p by n, input.
- R, complex matrix, size m by n, output.

Return Value

- none.

Description

$R := A \cdot B^*$.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices A, B and R must be conformant.
3. The input and output matrix views must not overlap.

Notes

vsip_Dmprodt_P

Calculate the product of a matrix and the transpose of a matrix.

Prototype

```
void vsip_Dmprodt_P(
                    const vsip_Dmview_P *A,
                    const vsip_Dmview_P *B,
                    const vsip_Dmview_P *R);
```

The following instances are supported:

vsip_mprodt_f

vsip_cmprodt_f

Parameters

- A, real or complex matrix, size m by p, input.
- B, real or complex matrix, size n by p, input.
- R, real or complex matrix, size m by n, output.

Return Value

- none.

Description

$R := A \cdot B^{T}$.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices A, B and R must be conformant.
3. The input and output matrix views must not overlap.

Notes

vsip_Dmvprod_P

Calculate a matrix–vector product.

Prototype

```
void vsip_Dmvprod_P(
                    const vsip_Dmview_P  *A,
                    const vsip_Dvview_P  *X,
                    const vsip_Dvview_P  *Y);
```

The following instances are supported:

vsip_mvprod_f

vsip_cmvprod_f

Parameters

- A, real or complex matrix, size m by n, input.
- X, real or complex vector, length n, input.
- Y, real or complex vector, length m, output.

Return Value

- none.

Description

$Y := A \cdot X$.

Restrictions

The result vector view may not overlap input matrix or vector views.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes

vsip_Dmtrans_P

Transpose a matrix.

Prototype

```
void vsip_Dmtrans_P(
                    const vsip_Dmview_P  *A,
                    const vsip_Dmview_P  *R);
```

The following instances are supported:

vsip_mtrans_f

vsip_cmtrans_f

Parameters

- A, real or complex matrix, size m by n, input.
- R, real or complex matrix, size n by m, output.

Return Value

- none.

Description

$R := A^T$.

Restrictions

If the matrix A is square, the transpose is in place if A and R resolve to the same object, otherwise A and R must be disjoint.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices A and R must be conformant.
3. If the matrix is not square, the input and output matrix views must not overlap. If the matrix is square, the input and output views must be identical views of the same block (in-place), or must not overlap.

Notes

vsip_Dvdot_P

Compute the inner (dot) product of two vectors.

Prototype

```
vsip_Dscalar_P vsip_Dvdot_P(
                    const vsip_Dvview_P  *A,
                    const vsip_Dvview_P  *B);
```

The following instances are supported:

vsip_vdot_f

vsip_cvdot_f

Parameters

- A, real or complex vector, length n, input.
- B, real or complex vector, length n, input.

Return Value

- real or complex scalar.

Description

return value := $A^T \cdot B$.

Restrictions

Overflow may occur.

Errors

The arguments must conform to the following:

1. Arguments for input must be the same size.
2. All view objects must be valid.

Notes

vsip_Dvmprod_P

Calculate a vector–matrix product.

Prototype

```
void vsip_Dvmprod_P(
                    const vsip_Dvview_P *X,
                    const vsip_Dmview_P *A,
                    const vsip_Dvview_P *Y);
```

The following instances are supported:

```
vsip_vmprod_f
vsip_cvmprod_f
```

Parameters

- X, real or complex vector, length m, input.
- A, real or complex matrix, size m by n, input.
- Y, real or complex vector, length n, output.

Return Value

- none.

Description

$Y := X^T \cdot A$.

Restrictions

The result vector view may not overlap the input matrix or vector views.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes

vsip_vouter_f

Calculate the outer product of two vectors.

Prototype

```
void vsip_vouter_f(
                     const vsip_scalar_f  alpha,
                     const vsip_vview_f  *X,
                     const vsip_vview_f  *Y,
                     const vsip_vview_f  *R);
```

Parameters

- alpha, real scalar, input.
- X, real vector, length m, input.
- Y, real vector, length n, input.
- R, real vector, size m by n, output.

Return Value

- none.

Description

R := alpha · X · Y^T.

Restrictions

The result matrix view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors and matrix must be conformant
3. The output matrix view and the input vector views must not overlap.

Notes

vsip_cvouter_f

Calculate the outer product of two vectors.

Prototype

```
void vsip_cvouter_f(
                     const vsip_cscalar_f  alpha,
                     const vsip_cvview_f  *X,
                     const vsip_cvview_f  *Y,
                     const vsip_cvview_f  *R);
```

Parameters

- alpha, complex scalar, input.
- X, complex vector, length m, input.
- Y, complex vector, length n, input.
- R, complex vector, size m by n, output.

Return Value

- none.

Description

R := alpha · X · Y^H.

Restrictions

The result matrix view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors and matrix must be conformant
3. The output matrix view and the input vector views must not overlap.

Notes

8.2 Special Linear System Solvers

- vsip_covsol_f
- vsip_ccovsol_f
- vsip_llsqsol_f
- vsip_cllsqsol_f
- vsip_toepsol_f
- vsip_ctoepsol_f

vsip_covsol_f

Solve a covariance linear system problem.

Prototype

```
int vsip_covsol_f(
                    const vsip_mview_f  *A,
                    const vsip_mview_f  *XB);
```

Parameters

- A, real matrix, size m by n, input. The matrix A.
- XB, real matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: A is not of full rank.

Description

Solves the covariance linear system problem $A^{\mathrm{T}}AX = B$ where A is an m by n matrix of rank n and B is an n by p matrix, and $n \leqslant m$.

Returns zero on success, -1 on memory allocation failure, and a positive value if A does not have full column rank.

Restrictions

The matrix A may be overwritten.

Errors

The input and output/input objects must conform to the following:

1. All objects must be valid.
2. The matrices A and XB must be conformant.

Notes

This function allocates and frees its own temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a covariance problem and they support explicit creation and destruction.

The matrix A is assumed to be of full rank. This property is checked and a positive return value indicates that an error occurred.

vsip_ccovsol_f

Solve a covariance linear system problem.

Prototype

```
int vsip_ccovsol_f(
                    const vsip_cmview_f  *A,
                    const vsip_cmview_f  *XB);
```

Parameters

- A, complex matrix, size m by n, input. The matrix A.
- XB, complex matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: A is not of full rank.

Description

Solves the covariance linear system problem $A^{\mathrm{H}}AX = B$ where A is an m by n matrix of rank n and B is an n by p matrix, and $n \leqslant m$.

Returns zero on success, -1 on memory allocation failure, and a positive value if A does not have full column rank.

Restrictions

The matrix A may be overwritten.

Errors

The input and output/input objects must conform to the following:

1. All objects must be valid.
2. The matrices A and XB must be conformant.

Notes

This function allocates and frees its own temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a covariance problem and they support explicit creation and destruction.

The matrix A is assumed to be of full rank. This property is checked and a positive return value indicates that an error occurred.

vsip_llsqsol_f

Solve a linear least squares problem.

Prototype

```
int vsip_llsqsol_f(
                    const vsip_mview_f  *A,
                    const vsip_mview_f  *XB);
```

Parameters

- A, real matrix, size m by n, input. The matrix A.
- XB, real matrix, size m by p, modified in place. On input, the matrix B. On output, the first n rows contain the matrix X.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: A is not of full rank.

Description

Solves the linear least squares problem $\min||AX - B||_2$ where A is an m by n matrix of rank n and B is an m by p matrix, and $n \leqslant m$.

Returns zero on success, -1 on memory allocation failure, and a positive value if A does not have full column rank.

Restrictions

The matrix A is overwritten.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrices A and XB must be conformant.

Notes

This function allocates and frees its own temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a covariance problem and they support explicit creation and destruction.

The matrix A is assumed to be of full rank. This property is checked. A positive return value indicates that the matrix did not have full column rank and the algorithm failed to be completed.

Since the output data length may be smaller than the input data length, it is recommended that a subview of the input vector be created which defines a vector view of the output data.

vsip_cllsqsol_f

Solve a linear least squares problem.

Prototype

```
int vsip_cllsqsol_f(
                    const vsip_cmview_f  *A,
                    const vsip_cmview_f  *XB);
```

Parameters

- A, complex matrix, size m by n, input. The matrix A.
- XB, complex matrix, size m by p, modified in place. On input, the matrix B. On output, the first n rows contain the matrix X.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: A is not of full rank.

Description

Solves the linear least squares problem $\min||AX - B||_2$ where A is an m by n matrix of rank n and B is an m by p matrix, and $n \leqslant m$.

Returns zero on success, -1 on memory allocation failure, and a positive value if A does not have full column rank.

Restrictions

The matrix A is overwritten.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrices A and XB must be conformant.

Notes

This function allocates and frees its own temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a covariance problem and they support explicit creation and destruction.

The matrix A is assumed to be of full rank. This property is checked. A positive return value indicates that the matrix did not have full column rank and the algorithm failed to be completed.

Since the output data length may be smaller than the input data length, it is recommended that a subview of the input vector be created which defines a vector view of the output data.

vsip_toepsol_f

Solve a real symmetric positive definite Toeplitz linear system.

Prototype

```
int vsip_toepsol_f(
                    const vsip_vview_f  *T,
                    const vsip_vview_f  *B,
                    const vsip_vview_f  *W,
                    const vsip_vview_f  *X);
```

Parameters

- T, real vector, length n, input. First row of the Toeplitz matrix T.
- B, real vector, length n, input. The vector B.
- W, real vector, length n, modified in place. Workspace.
- X, real vector, length n, output. The vector x.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: T is not positive definite.

Description

Solves the real symmetric positive definite Toeplitz linear system $Tx = B$ where

$$T = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-2} & t_{n-1} \\ t_1 & t_0 & t_1 & & t_{n-2} \\ \vdots & t_1 & \ddots & \ddots & \vdots \\ t_{n-2} & & \ddots & \ddots & t_1 \\ t_{n-1} & t_{n-2} & \cdots & t_1 & t_0 \end{bmatrix}.$$

We only need a vector containing the first row of T to specify the system.

Returns zero on success, -1 on memory allocation failure, and a positive value if T is not positive definite.

Restrictions

The result vector view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors t, x, w, and b must be conformant.
3. The input vector views and output vector view must not overlap.

Notes

The matrix T is assumed to be of full rank and positive definite. This property is not checked. A positive return value indicates that an error occurred and the algorithm failed to be completed.

vsip_ctoepsol_f

Solve a complex Hermitian positive definite Toeplitz linear system.

Prototype

```
int vsip_ctoepsol_f(
                    const vsip_cvview_f  *T,
                    const vsip_cvview_f  *B,
                    const vsip_cvview_f  *W,
                    const vsip_cvview_f  *X);
```

Parameters

- T, complex vector, length n, input. First row of the Toeplitz matrix T.
- B, complex vector, length n, input. The vector B.
- W, complex vector, length n, modified in place. Workspace.
- X, complex vector, length n, output. The vector x.

Return Value

- 0 : success.
- -1 : out of memory.
- $\geqslant 1$: T is not positive definite.

Description

Solves the complex Hermitian positive definite Toeplitz linear system $Tx = B$ where

$$T = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-2} & t_{n-1} \\ t_1^* & t_0 & t_1 & & t_{n-2} \\ \vdots & t_1^* & \ddots & \ddots & \vdots \\ t_{n-2}^* & & \ddots & \ddots & t_1 \\ t_{n-1}^* & t_{n-2}^* & \cdots & t_1^* & t_0 \end{bmatrix}.$$

We only need a vector containing the first row of T to specify the system.

Returns zero on success, -1 on memory allocation failure, and a positive value if T is not positive definite.

Restrictions

The result vector view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors t, x, w, and b must be conformant.
3. The input vector views and output vector view must not overlap.

Notes

The matrix T is assumed to be of full rank and positive definite. This property is not checked. A positive return value indicates that an error occurred and the algorithm failed to be completed.

8.3 General Square Linear System Solver

- vsip_Dlud_P
- vsip_Dlud_create_P
- vsip_Dlud_destroy_P
- vsip_Dlud_getattr_P
- vsip_lusol_f
- vsip_clusol_f

vsip_Dlud_P

Compute an LU decomposition of a square matrix using partial pivoting.

Prototype

```
int vsip_Dlud_P(
                    vsip_clu_P          *lud,
                    const vsip_Dmview_P *A);
```

The following instances are supported:

```
vsip_lud_f
vsip_clud_f
```

Parameters

- `lud`, structure, input.
- `A`, real or complex matrix, size n by n, modified in place.

Return Value

- Error code.

Description

Computes the LU decomposition of a general square matrix A using partial pivoting with row interchanges.

An LU decomposition is a factorisation of the form $A = PLU$ where P is a permutation matrix, L is lower triangular, and U is upper triangular.

Returns zero on success, and non-zero if A does not have full rank.

Restrictions

The matrix A is overwritten by the decomposition, and must not be modified as long as the factorisation is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix A and the LU decomposition object must be conformant.

Notes

The matrix A is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and a zero pivot element was encountered.

vsip_Dlud_create_P

Create an LU decomposition object.

Prototype

```
vsip_Dlu_P * vsip_Dlud_create_P(
                    const vsip_length N);
```

The following instances are supported:

```
vsip_lud_create_f
vsip_clud_create_f
```

Parameters

- N, integer scalar, input.

Return Value

- structure.

Description

Creates an LU decomposition object. The LU decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The LU decomposition object is used to compute the LU decomposition of a general square matrix A using partial pivoting with row interchanges.

An LU decomposition is a factorisation of the form $A = PLU$ where P is a permutation matrix, L is lower triangular, and U is upper triangular.

NULL is returned if the create fails.

Restrictions

Errors

The input parameter must conform to the following:

1. N is positive.

Notes

vsip_Dlud_destroy_P

Destroy an LU decomposition object.

Prototype

```
int vsip_Dlud_destroy_P(
                    vsip_Dlu_P *lud);
```

The following instances are supported:

```
vsip_lud_destroy_f
vsip_clud_destroy_f
```

Parameters

- `lud`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) an LU decomposition object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The input argument must conform to the following:

1. The LU decomposition object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_Dlud_getattr_P

Returns the attributes of an LU decomposition object.

Prototype

```
void vsip_Dlud_getattr_P(
                    const vsip_Dlu_P  *lud,
                    vsip_Dlu_attr_P   *attr);
```

The following instances are supported:

```
vsip_lud_getattr_f
vsip_clud_getattr_f
```

Parameters

- `lud`, structure, input.
- `attr`, pointer to structure, output.
  The attribute structure contains the following information:

`vsip_length`	`n`	number of rows and columns in matrix

Return Value

- none.

Description

Returns the attributes of an LU decomposition object.

Restrictions

Errors

The arguments must conform to the following:

1. The LU decomposition object `lud` must be valid.
2. The attribute pointer `attr` must not be `NULL`.

Notes

vsip_lusol_f

Solve a square linear system.

Prototype

```
int vsip_lusol_f(
                    const vsip_clu_f    *clud,
                    const vsip_mat_op    opA,
                    const vsip_mview_f  *XB);
```

Parameters

- clud, structure, input. An LU decomposition object for the matrix A.
- opA, enumerated type, input.

  VSIP_MAT_NTRANS no transformation
  VSIP_MAT_TRANS transpose
- XB, real matrix, size n by p, modified in place. (n is implicit, provided by clud.) On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solve the linear system $\mathrm{op}(A)X = B$ where op() is either the identity or transpose operation, for a general matrix A using the decomposition computed by vsip_lud_f

A is an n by n matrix of rank n and B is an n by p matrix.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix B and the LU decomposition object clud must be conformant.
3. opA must be valid.

Notes

It is safe to call this function after `vsip_lud_f`. fails. This will result in a non-zero unsuccessful return value.

vsip_clusol_f

Solve a square linear system.

Prototype

```
int vsip_clusol_f(
                    const vsip_clu_f     *clud,
                    const vsip_mat_op     opA,
                    const vsip_cmview_f  *XB);
```

Parameters

- clud, structure, input. An LU decomposition object for the matrix A.
- opA, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_HERM	Hermitian (conjugate transpose)

- XB, complex matrix, size n by p, modified in place. (n is implicit, provided by clud.) On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solve the linear system $\mathrm{op}(A)X = B$ where op() is either the identity or Hermitian operation, for a general matrix A using the decomposition computed by vsip_clud_f

A is an n by n matrix of rank n and B is an n by p matrix.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix B and the LU decomposition object clud must be conformant.
3. opA must be valid.

Notes

It is safe to call this function after `vsip_clud_f`. fails. This will result in a non-zero unsuccessful return value.

8.4 Symmetric Positive Definite Linear System Solver

- vsip_chold_f
- vsip_cchold_f
- vsip_chold_create_f
- vsip_cchold_create_f
- vsip_Dchold_destroy_P
- vsip_Dchold_getattr_P
- vsip_cholsol_f
- vsip_ccholsol_f

vsip_chold_f

Compute a Cholesky decomposition of a symmetric positive definite matrix.

Prototype

```
int vsip_chold_f(
                    vsip_cchol_f       *chold,
                    const vsip_mview_f *A);
```

Parameters

- `chold`, structure, input.
- `A`, real matrix, size n by n, modified in place.

Return Value

- Error code.

Description

The Cholesky decomposition of a symmetric positive definite n by n matrix A is given by $A = LL^{\mathrm{T}}$ where L is a lower triangular matrix.

There is not a utility function for accessing the factors.

Returns zero on success. The routine will fail if a leading minor is not positive definite.

Restrictions

The matrix A is overwritten by the decomposition and must not be modified as long as the decomposition is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix `A` and the Cholesky decomposition object `chold` must be conformant.

Notes

The matrix A is assumed to be symmetric. This property is not checked. There is no reduced storage format for symmetric matrices so the full matrix must be specified. However, only half the matrix is referenced and modified.

vsip_cchold_f

Compute a Cholesky decomposition of a Hermitian positive definite matrix.

Prototype

```
int vsip_cchold_f(
                    vsip_cchol_f        *chold,
                    const vsip_cmview_f *A);
```

Parameters

- `chold`, structure, input.
- `A`, complex matrix, size n by n, modified in place.

Return Value

- Error code.

Description

The Cholesky decomposition of a Hermitian positive definite n by n matrix A is given by $A = LL^{\mathrm{H}}$ where L is a lower triangular matrix.

There is not a utility function for accessing the factors.

Returns zero on success. The routine will fail if a leading minor is not positive definite.

Restrictions

The matrix A is overwritten by the decomposition and must not be modified as long as the decomposition is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix `A` and the Cholesky decomposition object `chold` must be conformant.

Notes

The matrix A is assumed to be Hermitian. This property is not checked. There is no reduced storage format for Hermitian matrices so the full matrix must be specified. However, only half the matrix is referenced and modified.

vsip_chold_create_f

Creates a Cholesky decomposition object.

Prototype

```
vsip_chol_f * vsip_chold_create_f(
                    const vsip_mat_uplo uplo,
                    const vsip_length   n);
```

Parameters

- uplo, enumerated type, input.

VSIP_TR_LOW	lower triangle
VSIP_TR_UPP	upper triangle

- n, integer scalar, input.

Return Value

- structure.

Description

Creates a Cholesky decomposition object. The Cholesky decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The Cholesky decomposition object is used to compute the Cholesky decomposition of a symmetric positive definite n by n matrix A.

The Cholesky decomposition of a symmetric positive definite n by n matrix A is given by $A = LL^{\mathrm{T}}$ where L is a lower triangular matrix.

There is not a utility function for accessing the factors.

NULL is returned if the create fails.

Restrictions

Errors

The input parameters must conform to the following:

1. n is positive.
2. uplo is valid.

Notes

vsip_cchold_create_f

Creates a Cholesky decomposition object.

Prototype

```
vsip_cchol_f * vsip_cchold_create_f(
                    const vsip_mat_uplo uplo,
                    const vsip_length   n);
```

Parameters

- uplo, enumerated type, input.

VSIP_TR_LOW	lower triangle
VSIP_TR_UPP	upper triangle

- n, integer scalar, input.

Return Value

- structure.

Description

Creates a Cholesky decomposition object. The Cholesky decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The Cholesky decomposition object is used to compute the Cholesky decomposition of a Hermitian positive definite n by n matrix A.

The Cholesky decomposition of a Hermitian positive definite n by n matrix A is given by $A = LL^{\mathrm{H}}$ where L is a lower triangular matrix.

There is not a utility function for accessing the factors.

NULL is returned if the create fails.

Restrictions

Errors

The input parameters must conform to the following:

1. n is positive.
2. uplo is valid.

Notes

vsip_Dchold_destroy_P

Destroy a Cholesky decomposition object.

Prototype

```
int vsip_Dchold_destroy_P(
                    vsip_Dchol_P *chold);
```

The following instances are supported:

```
vsip_chold_destroy_f
vsip_cchold_destroy_f
```

Parameters

- `chold`, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) a Cholesky decomposition object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The input object must conform to the following:

1. The Cholesky decomposition object must be valid. An argument of `NULL` is not an error.

Notes

An argument of `NULL` is not an error.

vsip_Dchold_getattr_P

Returns the attributes of a Cholesky decomposition object.

Prototype

```
void vsip_Dchold_getattr_P(
                    const vsip_Dchol_P  *chold,
                    vsip_Dchol_attr_P   *attr);
```

The following instances are supported:

```
vsip_chold_getattr_f
vsip_cchold_getattr_f
```

Parameters

- `chold`, structure, input.
- `attr`, pointer to structure, output.
  The attribute structure contains the following information:

`vsip_mat_uplo`	`uplo`	upper or lower triangular matrix
`vsip_length`	`n`	number of rows and columns in matrix

Return Value

- none.

Description

Returns the attributes of a Cholesky decomposition object.

Restrictions

Errors

The input and output arguments must conform to the following:

1. The Cholesky decomposition object `chold` must be valid.
2. The attribute pointer `attr` must not be `NULL`.

Notes

vsip_cholsol_f

Solve a symmetric positive definite linear system.

Prototype

```
int vsip_cholsol_f(
                    const vsip_cchol_f  *chold,
                    const vsip_mview_f  *XB);
```

Parameters

- chold, structure, input. A Cholesky decomposition object for the matrix A.
- XB, real matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solve the linear system $AX = B$ for a symmetric positive definite matrix A using the decomposition computed by vsip_chold_f.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix XB and the Cholesky decomposition object chold must be conformant.

Notes

It is safe to call this function after vsip_chold_f fails. This will result in a non-zero unsuccessful return value.

vsip_ccholsol_f

Solve a Hermitian positive definite linear system.

Prototype

```
int vsip_ccholsol_f(
                    const vsip_cchol_f   *chold,
                    const vsip_cmview_f  *XB);
```

Parameters

- `chold`, structure, input. A Cholesky decomposition object for the matrix A.
- `XB`, complex matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solve the linear system $AX = B$ for a Hermitian positive definite matrix A using the decomposition computed by `vsip_cchold_f`.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix `XB` and the Cholesky decomposition object `chold` must be conformant.

Notes

It is safe to call this function after `vsip_cchold_f` fails. This will result in a non-zero unsuccessful return value.

8.5 Overdetermined Linear System Solver

- vsip_qrd_f
- vsip_cqrd_f
- vsip_qrd_create_f
- vsip_cqrd_create_f
- vsip_Dqrd_destroy_P
- vsip_Dqrd_getattr_P
- vsip_qrdprodq_f
- vsip_cqrdprodq_f
- vsip_qrdsolr_f
- vsip_cqrdsolr_f
- vsip_qrsol_f
- vsip_cqrsol_f

vsip_qrd_f

Compute a QR decomposition of a matrix .

Prototype

```
int vsip_qrd_f(
                    vsip_cqr_f          *qrd,
                    const vsip_mview_f  *A);
```

Parameters

- qrd, structure, input.
- A, real matrix, size m by n, modified in place.

Return Value

- Error code.

Description

Computes the QR decomposition of an m by n matrix A where $n \leqslant m$.

The QR decomposition of A is given by $A = QR$ where Q is an m by n orthogonal matrix ($Q^{\mathrm{T}}Q = I$) and R is an n by n upper triangular matrix. If A has full rank then R is non-singular. The routine does not perform any column interchanges.

Returns zero on success. It will fail if A does not have full column rank.

Restrictions

The matrix A is overwritten by the decomposition and must not be modified as long as the factorisation is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix A and the QR decomposition object must be conformant.

Notes

The matrix A is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and a zero diagonal element of R was encountered.

vsip_cqrd_f

Compute a QR decomposition of a matrix .

Prototype

```
int vsip_cqrd_f(
                    vsip_cqr_f           *qrd,
                    const vsip_cmview_f  *A);
```

Parameters

- qrd, structure, input.
- A, complex matrix, size m by n, modified in place.

Return Value

- Error code.

Description

Computes the QR decomposition of an m by n matrix A where $n \leqslant m$.

The QR decomposition of A is given by $A = QR$ where Q is an m by n unitary matrix ($Q^{\mathrm{H}}Q = I$) and R is an n by n upper triangular matrix. If A has full rank then R is non-singular. The routine does not perform any column interchanges.

Returns zero on success. It will fail if A does not have full column rank.

Restrictions

The matrix A is overwritten by the decomposition and must not be modified as long as the factorisation is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix A and the QR decomposition object must be conformant.

Notes

The matrix A is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and a zero diagonal element of R was encountered.

vsip_qrd_create_f

Create a QR decomposition object.

Prototype

```
vsip_qr_f * vsip_qrd_create_f(
                    const vsip_length   m,
                    const vsip_length   n,
                    const vsip_qrd_qopt qopt);
```

Parameters

- m, integer scalar, input.
- n, integer scalar, input.
- qopt, enumerated type, input.

VSIP_QRD_NOSAVEQ	do not save Q
VSIP_QRD_SAVEQ	save full Q
VSIP_QRD_SAVEQ1	save skinny Q

Return Value

- structure.

Description

Creates a QR decomposition object. The QR decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The QR decomposition of A is given by $A = QR$ where Q is an m by n orthogonal matrix ($Q^{\mathrm{T}}Q = I$) and R is an n by n upper triangular matrix, and $n \leqslant m$. If A has full rank then R is non-singular.

The matrix R will be generated and retained for later use. There is a flag to indicate whether Q is retained and, if so, in what format.

NULL is returned if the create fails.

Restrictions

Errors

The input arguments must conform to the following:

1. m and n positive with n $\leqslant$ m

2. `qopt` is valid.

Notes

vsip_cqrd_create_f

Create a QR decomposition object.

Prototype

```
vsip_cqr_f * vsip_cqrd_create_f(
                    const vsip_length   m,
                    const vsip_length   n,
                    const vsip_qrd_qopt qopt);
```

Parameters

- m, integer scalar, input.
- n, integer scalar, input.
- qopt, enumerated type, input.

VSIP_QRD_NOSAVEQ	do not save Q
VSIP_QRD_SAVEQ	save full Q
VSIP_QRD_SAVEQ1	save skinny Q

Return Value

- structure.

Description

Creates a QR decomposition object. The QR decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The QR decomposition of A is given by $A = QR$ where Q is an m by n unitary matrix ($Q^{\mathrm{H}}Q = I$) and R is an n by n upper triangular matrix, and $n \leqslant m$. If A has full rank then R is non-singular.

The matrix R will be generated and retained for later use. There is a flag to indicate whether Q is retained and, if so, in what format.

NULL is returned if the create fails.

Restrictions

Errors

The input arguments must conform to the following:

1. m and n positive with n $\leqslant$ m

2. `qopt` is valid.

Notes

vsip_Dqrd_destroy_P

Destroy a QR decomposition object.

Prototype

```
int vsip_Dqrd_destroy_P(
                    vsip_Dqr_P *qrd);
```

The following instances are supported:

```
vsip_qrd_destroy_f
vsip_cqrd_destroy_f
```

Parameters

- qrd, structure, input.

Return Value

- Error code.

Description

Destroys (frees the memory used by) a QR decomposition object. Returns zero on success, non-zero on failure.

Restrictions

Errors

The input object must conform to the following:

1. The QR decomposition object must be valid. An argument of NULL is not an error.

Notes

An argument of NULL is not an error.

vsip_Dqrd_getattr_P

Returns the attributes of a QR decomposition object.

Prototype

```
void vsip_Dqrd_getattr_P(
                    const vsip_Dqr_P  *qrd,
                    vsip_Dqr_attr_P   *attr);
```

The following instances are supported:

```
vsip_qrd_getattr_f
vsip_cqrd_getattr_f
```

Parameters

- qrd, structure, input.
- attr, pointer to structure, output.
  The attribute structure contains the following information:

vsip_length	m	number of rows in input matrix
vsip_length	n	number of columns in input matrix
vsip_qrd_qopt	Qopt	matrix Q is saved/not saved

Return Value

- none.

Description

Returns the attributes of a QR decomposition object.

Restrictions

Errors

The arguments must conform to the following:

1. The QR decomposition object qrd must be valid.
2. The attribute pointer attr must not be NULL.

Notes

vsip_qrdprodq_f

Multiply a matrix by the matrix Q from a QR decomposition.

Prototype

```
int vsip_qrdprodq_f(
                    const vsip_qr_f      *qrd,
                    const vsip_mat_op     opQ,
                    const vsip_mat_side   apQ,
                    const vsip_mview_f   *C);
```

Parameters

- qrd, structure, input.
- opQ, enumerated type, input.

  VSIP_MAT_NTRANS no transformation
  VSIP_MAT_TRANS transpose
- apQ, enumerated type, input.

  VSIP_MAT_LSIDE left side
  VSIP_MAT_RSIDE right side
- C, real matrix, size p by q, modified in place.

Return Value

- Error code.

Description

This function overwrites a matrix C with the product $\mathrm{op}(Q) \cdot C$ if multiplied on the left, or $C \cdot \mathrm{op}(Q)$ if multiplied on the right. The matrix Q is computed by vsip_qrd_f (its size depends on which option was used to store it), and op() is either the identity or transpose operation.

In some cases, the output matrix is larger than C.

Returns zero on success, non-zero on failure.

Restrictions

Since the output data space may be larger than the input data space, it is required that the input allows storage in the block for the output data. This means the row stride and column stride must be calculated to accommodate the larger data space, whether it be input or output.

Errors

The arguments must conform to the following:

1. `opQ` is valid
2. `apQ` is valid
3. The matrix `C` and the QR decomposition object `qrd` must be conformant.
4. The QR decomposition object must have specified retaining the Q matrix when it was created.

Notes

It is safe to call this function after `vsip_qrd_f`. fails. This will result in a non-zero unsuccessful return value.

If the output data space is larger than the input data space create a matrix view large enough to hold the output data. Create a subview of this with index offset at (0,0) of proper size to hold the input data. The new (sub) view is then the input to the function, and the original view will hold the output data.

vsip_cqrdprodq_f

Multiply a matrix by the matrix Q from a QR decomposition.

Prototype

```
int vsip_cqrdprodq_f(
                    const vsip_cqr_f     *qrd,
                    const vsip_mat_op     opQ,
                    const vsip_mat_side   apQ,
                    const vsip_cmview_f  *C);
```

Parameters

- qrd, structure, input.
- opQ, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose
VSIP_MAT_HERM	Hermitian (conjugate transpose)

- apQ, enumerated type, input.

VSIP_MAT_LSIDE	left side
VSIP_MAT_RSIDE	right side

- C, complex matrix, size p by q, modified in place.

Return Value

- Error code.

Description

This function overwrites a matrix C with the product $\mathrm{op}(Q) \cdot C$ if multiplied on the left, or $C \cdot \mathrm{op}(Q)$ if multiplied on the right. The matrix Q is computed by vsip_cqrd_f (its size depends on which option was used to store it), and op() is either the identity or (identity, transpose, or Hermitian). operation.

In some cases, the output matrix is larger than C.

Returns zero on success, non-zero on failure.

Restrictions

Since the output data space may be larger than the input data space, it is required that the input allows storage in the block for the output data. This means the row stride and

column stride must be calculated to accommodate the larger data space, whether it be input or output.

Errors

The arguments must conform to the following:

1. `opQ` is valid
2. `apQ` is valid
3. The matrix `C` and the QR decomposition object `qrd` must be conformant.
4. The QR decomposition object must have specified retaining the Q matrix when it was created.

Notes

It is safe to call this function after `vsip_cqrd_f`. fails. This will result in a non-zero unsuccessful return value.

If the output data space is larger than the input data space create a matrix view large enough to hold the output data. Create a subview of this with index offset at (0,0) of proper size to hold the input data. The new (sub) view is then the input to the function, and the original view will hold the output data.

vsip_qrdsolr_f

Solve linear system based on the matrix R, from QR decomposition of the matrix A.

Prototype

```
int vsip_qrdsolr_f(
                    const vsip_qr_f      *qrd,
                    const vsip_mat_op     OpR,
                    const vsip_scalar_f   alpha,
                    const vsip_mview_f   *XB);
```

Parameters

- qrd, structure, input. A QR decomposition object for the matrix A.
- OpR, enumerated type, input.

  VSIP_MAT_NTRANS no transformation
  VSIP_MAT_TRANS transpose
- alpha, real scalar, input.
- XB, real matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solves a triangular linear system of the form $\mathrm{op}(R)X = \alpha B$ where op() is either the identity or transpose operation, using the decomposition computed by vsip_qrd_f.

R is an n by n upper triangular matrix; X and B are n by p matrices.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. All of the objects must be valid.
2. OpR is valid.

3. The matrix XB and the QR decomposition object qrd must be conformant.

Notes

It is safe to call this function after vsip_qrd_f. fails. This will result in a non-zero unsuccessful return value.

vsip_cqrdsolr_f

Solve linear system based on the matrix R, from QR decomposition of the matrix A.

Prototype

```
int vsip_cqrdsolr_f(
                    const vsip_cqr_f      *qrd,
                    const vsip_mat_op      OpR,
                    const vsip_cscalar_f   alpha,
                    const vsip_cmview_f   *XB);
```

Parameters

- qrd, structure, input. A QR decomposition object for the matrix A.
- OpR, enumerated type, input.

VSIP_MAT_NTRANS	no transformation
VSIP_MAT_TRANS	transpose
VSIP_MAT_HERM	Hermitian (conjugate transpose)

- alpha, complex scalar, input.
- XB, complex matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Solves a triangular linear system of the form $\mathrm{op}(R)X = \alpha B$ where op() is either the identity or (identity, transpose, or Hermitian). operation, using the decomposition computed by vsip_cqrd_f.

R is an n by n upper triangular matrix; X and B are n by p matrices.

Returns zero on success, non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. All of the objects must be valid.

2. OpR is valid.
3. The matrix XB and the QR decomposition object qrd must be conformant.

Notes

It is safe to call this function after vsip_cqrd_f. fails. This will result in a non-zero unsuccessful return value.

vsip_qrsol_f

Solve either a linear covariance or linear least squares problem.

Prototype

```
int vsip_qrsol_f(
                    const vsip_qr_f      *qrd,
                    const vsip_qrd_prob   prob,
                    const vsip_mview_f   *XB);
```

Parameters

- qrd, structure, input. A QR decomposition object for the matrix A.
- prob, enumerated type, input.

  VSIP_COV solve a covariance linear system problem
  VSIP_LLS solve a linear least squares problem
- XB, real matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Let A be an m by n matrix of rank n with $n \leqslant m$, and let B be a matrix of size n by p for a covariance problem or m by p for a least squares problem.

This routine solves one of the following problems using the decomposition computed by vsip_qrd_f

- a covariance linear system problem $A^{\mathrm{T}}AX = B$
- a linear least squares problem $\min|AX - B|_2$.

Restrictions

This routine will fail if A does not have full rank.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix `XB` and the QR decomposition object `qrd` must be conformant.
3. `prob` is valid.

Notes

It is safe to call this function after `vsip_qrd_f` fails. This will result in a non-zero unsuccessful return value.

vsip_cqrsol_f

Solve either a linear covariance or linear least squares problem.

Prototype

```
int vsip_cqrsol_f(
                    const vsip_cqr_f     *qrd,
                    const vsip_qrd_prob   prob,
                    const vsip_cmview_f  *XB);
```

Parameters

- qrd, structure, input. A QR decomposition object for the matrix A.
- prob, enumerated type, input.

  VSIP_COV solve a covariance linear system problem
  VSIP_LLS solve a linear least squares problem
- XB, complex matrix, size n by p, modified in place. On input, the matrix B. On output, the matrix X.

Return Value

- Error code.

Description

Let A be an m by n matrix of rank n with $n \leqslant m$, and let B be a matrix of size n by p for a covariance problem or m by p for a least squares problem.

This routine solves one of the following problems using the decomposition computed by vsip_qrd_f

- a covariance linear system problem $A^{\mathrm{H}}AX = B$
- a linear least squares problem $\min|AX - B|_2$.

Restrictions

This routine will fail if A does not have full rank.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix `XB` and the QR decomposition object `qrd` must be conformant.
3. `prob` is valid.

Notes

It is safe to call this function after `vsip_cqrd_f` fails. This will result in a non-zero unsuccessful return value.

Chapter 9. Using Routines in the VSIPL library

9.1 Header File.

The VSIPL function and object definitions are defined in a single include file called **vsipl.h** which is supplied within the include directory of each VSIPL distribution. The user must include this file in their VSIPL C source code in order to comile their code.

9.2 VSIPL Libraries.

The following two statically compiled libraries are supplied with each VSIPL distribution in the libs directory:

Filename	Library type	Description
libDvsip_c.a	Development VSIPL library	VSIPL library containing error checking.
libvsip_c.a	Performance VSIPL library	VSIPL library without error checking.

The user may link their VSIPL program with the development library for the purposes of software validation. The development library performs error checking with the function arguments. If any errors are detected the library will output an appropriate error message and terminate. Once the user's code has been validated with the development library and there are no coding problems found then they may link their code to the performance library. This library is faster because it does not perform any argument error checking.

9.3 Source Code Examples

This section includes a small number of simple C source code examples that show how VSIPL routines are called. The examples are made up of the following:

- **1D FFT Benchark Code:** Example code containing complex-to-real, real-to-complex and complex-to-complex VSIPL 1D FFT functions.
- **Complex Vector Multiply Benchmark Code:** Example code containing the complex vector multiply processing function **vsip_cvmul_f**.
- **Vector Sine Benchmark Code:** Example code containing the vector sine processing function **vsip_vsin_f**.
- **Multiple FFT Benchmark Code:** Example code containing the complex-to-complex multiple 1D FFT function **vsip_ccfftmip_f**.
- **Scalar and Vector Benchmark Code:** Example code containing the scalar/vector processing function **vsip_svmul_f** and vector processsing function **vsip_vadd_f**.

9.3.1 1D FFT Benchmark Code

The following code segment shows how VSIPL blocks/views are created/initialised in order to benchmark the processing functions **vsip_rcfftop_f**, **vsip_crfftop_f**, **vsip_ccfftop_f** and **vsip_ccfftip_f**.

```
/*****************************************************************************/
/*                                                                           */
/*                    benchfft()                                             */
/*                                                                           */
/* This benchmark function times a series of 1D VSIPL FFT routines and       */
/* outputs the average processing time in microseconds.                      */
/*                                                                           */
/* Functions being timed:                                                    */
/*     vsip_rcfftop_f - real-to-complex out-of-place 1D FFT routine;         */
/*     vsip_crfftop_f - complex-to-real out-of-place 1D FFT routine;         */
/*     vsip_ccfftop_f - complex-to-complex out-of-place 1D FFT routine;      */
/*     vsip_ccfftip_f - complex-to-complex in-place 1D FFT routine;          */
/*                                                                           */
/* Processing lengths:                                                       */
/*     Vector length of 8 cells to 1M of cells.                              */
/*                                                                           */
```

```
/*********************************************************************/
int benchfft(void)
{
   vsip_fft_f* fftrcop;
   vsip_fft_f* fftcrop;
   vsip_fft_f* fftccop;
   vsip_fft_f* fftccip;

   vsip_vview_f*  aa;
   vsip_cvview_f* bb;
   vsip_cvview_f* cc;
   vsip_cvview_f* dd;
   vsip_cvview_f* ee;
   vsip_cscalar_f z;
   vsip_cvattr_f  bb1;
   vsip_scalar_i  cycles;
   vsip_scalar_i  N, i, k;

   double tm, tm2;

   vsip_init(NULL);

   nasprintf("   N      rcfftop    crfftop      ccfftop      ccfftip\n");

   N = 8;
   while (N <= 1024*1024)
   {
      nasprintf("%7d     ", N);

      aa = vsip_vcreate_f(N, VSIP_MEM_NONE);


/*---------------RCFFTOP-----------------------------------------------*/
      bb = vsip_cvcreate_f(N/2 + 1, VSIP_MEM_NONE);

      vsip_vramp_f(0.0, 1.0, aa);

      fftrcop = vsip_rcfftop_create_f(N, 1.0, 1, VSIP_ALG_TIME);

      cycles = 100;

      tm2 = my_dtime();
      vsip_rcfftop_f(fftrcop, aa, bb);
      tm2 = my_dtime();
```

```
      for(i=0; i < cycles; i++)
      {
         vsip_rcfftop_f(fftrcop, aa, bb);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f  ", (double)tm);

      vsip_fft_destroy_f(fftrcop);

/*---------------CRFFTOP-----------------------------------------------------*/

      vsip_vramp_f(0.0, 0.0, aa);

      cycles = 100;

      fftcrop = vsip_crfftop_create_f(N, 1.0/N, 1, VSIP_ALG_TIME);

      tm = my_dtime();

      for(i=0; i < cycles; i++)
      {
         vsip_crfftop_f(fftcrop, bb, aa);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f     ", (double)tm);

      vsip_fft_destroy_f(fftcrop);

      vsip_cvalldestroy_f(bb);


/*---------------CCFFTOP-----------------------------------------------------*/

      cc = vsip_cvcreate_f(N, VSIP_MEM_NONE);
      dd = vsip_cvcreate_f(N, VSIP_MEM_NONE);

      if ((cc == NULL) || (dd == NULL))
      {
         nasprintf("vsip_cvcreate_f buffer creation failed!\n");
         return (0);
      }

      for(i = 0; i < N; i++)
```

```
      {
         z = vsip_cmplx_f( i + 1, -(i + 1));
         vsip_cvput_f(cc, i, z);
      }
      fftccop = vsip_ccfftop_create_f(N, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_TIME);
      cycles = 100;
      tm = my_dtime();

      for(i=0; i < cycles; i++)
      {
         vsip_ccfftop_f(fftccop, cc, dd);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f     ", (double)tm); fflush(stdout);

      vsip_fft_destroy_f(fftccop);

      vsip_cvalldestroy_f(dd);
      vsip_cvalldestroy_f(cc);

/*---------------CCFFTIP-FORWARD-----------------------------------------*/

      ee = vsip_cvcreate_f(N, VSIP_MEM_NONE);

      for(i = 0; i < N; i++)
      {
         z = vsip_cmplx_f( 0, 0);
         vsip_cvput_f(ee, i, z);
      }
      cycles = 100;
      fftccip = vsip_ccfftip_create_f(N, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_TIME);

      tm = my_dtime();

      for (k = 0; k < cycles; k++)
      {
         vsip_ccfftip_f(fftccip, ee);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f", (double)tm); fflush(stdout);

      vsip_fft_destroy_f(fftccip);
```

```
/*---------------CCFFTIP-INVERSE-----------------------------------------*/
#ifdef TEST_INVERSE
      fftccip = vsip_ccfftip_create_f(N, 1.0/N, VSIP_FFT_INV, 1, VSIP_ALG_TIME);

      tm = my_dtime();

      for(k = 0; k < cycles; k++)
         vsip_ccfftip_f(fftccip, ee);

      tm = my_dtime()/cycles;
      nasprintf("ccfftip inv 1.0 - %8.3f us\n\n", (double)tm);

      vsip_fft_destroy_f(fftccip);

      vsip_cvalldestroy_f(ee);
#endif
/*-------------------------------------------------------------------------*/

      vsip_valldestroy_f(aa);

       N = N * 2;
   }  /* end of while loop */

   vsip_finalize(NULL);

   return 0;

} /* End of benchfft.                                                      */
```

9.3.2 Complex Vector Multiply Benchmark Code.

The following code segment shows how VSIPL blocks/views are created/initialised in order to benchmark the processing function **vsip_cvmul_f**.

```
/*************************************************************************/
/*                                                                       */
/*                    benchcvmul()                                       */
/*                                                                       */
/* This benchmark function times a the complex vector multiply VSIPL     */
/* function vsip_cvmul_f and outputs the average processing time in      */
/* microseconds.                                                         */
/*                                                                       */
/* Processing lengths:                                                   */
/*     Vector length of 8 cells to 1M of cells.                          */
/*                                                                       */
/*************************************************************************/
int benchcvmul(void)
{
   vsip_scalar_i cycles, i, k, N;
   vsip_cvview_f* aa;
   vsip_cvview_f* bb;
   vsip_cvview_f* rr;
   vsip_cscalar_f z;

   double tm, tm2;

   vsip_init(NULL);

   nasprintf("   N      cvmulop     cvmulip                          \n");

   N = 8;
   while (N <= 1024*1024)
   {
      nasprintf("%7d     ", N);


/*---------------CVMULOP-------------------------------------------------*/
      aa = vsip_cvcreate_f(N, VSIP_MEM_NONE);
      bb = vsip_cvcreate_f(N, VSIP_MEM_NONE);
      rr = vsip_cvcreate_f(N, VSIP_MEM_NONE);

      for(i = 0; i < N; i++)
      {
         z = vsip_cmplx_f( i + 1, -(i + 1));
```

```
         vsip_cvput_f(aa, i, z);
         vsip_cvput_f(bb, i, z);
      }

      cycles = 1000;

      tm2 = my_dtime();

      for(i=0; i < cycles; i++)
      {
         vsip_cvmul_f(aa, bb, rr);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f  ", (double)tm);


/*---------------CVMULIP-----------------------------------------------------*/

      for(i = 0; i < N; i++)
      {
         z = vsip_cmplx_f( i + 1, -(i + 1));
         vsip_cvput_f(aa, i, z);
         vsip_cvput_f(bb, i, z);
      }


      cycles = 1000;

      tm2 = my_dtime();

      for(i=0; i < cycles; i++)
      {
         vsip_cvmul_f(aa, bb, bb);
      }

      tm = my_dtime()/cycles;
      nasprintf("%8.3f  \n", (double)tm);

      vsip_cvalldestroy_f(aa);
      vsip_cvalldestroy_f(bb);
      vsip_cvalldestroy_f(rr);

      N = N * 2;
   }  /* end of while loop */
```

```
   vsip_finalize(NULL);

   return 0;

} /* End of benchcvmul.                                                    */

/*************************************************************************/
```

9.3.3 Vector Sine Benchmark Code.

The following code segment shows how VSIPL blocks/views are created/initialised in order to benchmark the processing function **vsip_vsin_f**.

```
/*************************************************************************/
/*                                                                       */
/*                    time_vsin()                                        */
/*                                                                       */
/* This benchmark function times a the float vector sine VSIPL           */
/* function vsip_vsin_f and outputs the average processing time in       */
/* microseconds.                                                         */
/*                                                                       */
/* Processing lengths:                                                   */
/*     User given range of processing lengths.                           */
/*                                                                       */
/*************************************************************************/

vsip_scalar_i time_vsin(void)
{
   vsip_scalar_i index, test_no, v_length, time_loop;
   vsip_block_f *buff_block_out = NULL;
   vsip_vview_f *buff_view_out  = NULL;
   vsip_block_f *buff_block_in  = NULL;
   vsip_vview_f *buff_view_in   = NULL;
   vsip_scalar_f clock_freq;
   vsip_scalar_f ticks;
   time_val startTime;

   /* Set the timer clock frequency.                                     */
   clock_freq = get_clock_frequency();


   /* Initialise the board and library.                                  */
   init_routine();



   printf(" INPUT VECTOR  | OUTPUT VECTOR | TIMING.\n");
   printf(" Length        | Length        | Seconds.\n");
   printf("-----------------------------------------\n");
   for(test_no=0; test_no<N_TESTS; test_no++)
   {
      /* Define test dimensions.                                         */
      v_length = test_buffer[test_no].v_length;
```

```
printf(" %06d        | %06d        | ", v_length, v_length);


/* Create data block and vector view for the VSIPL tests.           */
buff_block_out = vsip_blockcreate_f((size_t)(v_length), 0);
if(buff_block_out == NULL)
{
   printf("Block creation error!\n");
   return 0;
}
buff_view_out = vsip_vbind_f(buff_block_out, 0, 1, v_length);
if(buff_view_out == NULL)
{
   printf("View creation error!\n");
   return 0;
}
buff_block_in = vsip_blockcreate_f((size_t)(v_length), 0);
if(buff_block_in == NULL)
{
   printf("Block creation error!\n");
   return 0;
}
buff_view_in = vsip_vbind_f(buff_block_in, 0, 1, v_length);
if(buff_view_in == NULL)
{
   printf("View creation error!\n");
   return 0;
}



/*********************************************************************
    Perform timing over N_ITS iterations.
 *********************************************************************/

/* Initialise the vsipl buffer.                                      */
set_vector_data(buff_view_in, v_length);


/* Start the timer...                                                */
startTime = get_cycles();

for(time_loop=0; time_loop<N_ITS; time_loop++)
```

```
      {
         /* Reverse order of vector by calling vsip_vsin_f.              */
         vsip_vsin_f(buff_view_in, buff_view_out);

      } /* End of timing loop.                                           */

      /* End the timer...                                               */
      ticks = (vsip_scalar_f)(get_cycles() - startTime);

      /* Output timings...                                              */
      printf("%g \n", (ticks/N_ITS) / clock_freq);


      /* Free up all views, blocks and plans.                           */
      vsip_vdestroy_f(buff_view_out);
      vsip_vdestroy_f(buff_view_in);
   }

   return 1;

} /* End of time_vsin.                                                 */
```

9.3.4 Multiple complex-to-complex FFT Benchmark Code.

The following code segment shows how VSIPL blocks/views are created/initialised in order to benchmark the processing function **vsip_ccfftmip_f**.

```
/*************************************************************************/
/*                                                                       */
/*                    time_multiple_fft()                                */
/*                                                                       */
/* This benchmark function times a the complex-to-complex VSIPL          */
/* function vsip_ccfftmip_f and outputs the average processing time in   */
/* microseconds.                                                         */
/*                                                                       */
/* Processing lengths:                                                   */
/*     User given range of processing lengths.                           */
/*                                                                       */
/*************************************************************************/
vsip_scalar_i time_multiple_fft(void)
{
   vsip_scalar_i test_no, fft_length, no_of_ffts, time_loop;
   vsip_fftm_f *buff_fft_plan = NULL;
   vsip_cblock_f *buff_block  = NULL;
   vsip_cmview_f *buff_view   = NULL;
   vsip_scalar_f clock_freq;
   vsip_scalar_f ticks;
   time_val startTime;

   /* Set the timer clock frequency.                                     */
   clock_freq = get_clock_frequency();


   /* Initialise the board and library.                                  */
   init_routine();


   printf(" MATRIX SIZE   |    FFT INFO      | TIMING.\n");
   printf(" Length Width  | Direction Type    | Seconds.\n");
   printf("-----------------------------------------------\n");
   for(test_no=0; test_no<N_TESTS; test_no++)
   {
      /* Define test dimensions.                                         */
      fft_length = test_buffer[test_no].fft_length;
      no_of_ffts = test_buffer[test_no].no_of_ffts;

      printf(" %06d %06d | forward ip rows  | ", fft_length, no_of_ffts);
```

```
/* Create data block and vector view for the VSIPL tests.          */
buff_block = vsip_cblockcreate_f((size_t)(fft_length * no_of_ffts), 0);
if(buff_block == NULL)
{
   printf("Block creation error!\n");
   return 0;
}
buff_view = vsip_cmbind_f(buff_block, 0, fft_length, no_of_ffts,
                          1, fft_length);
if(buff_view == NULL)
{
   printf("View creation error!\n");
   return 0;
}



/* FFT Plan Creation:                                              */
buff_fft_plan = vsip_ccfftmip_create_f((vsip_length)no_of_ffts,
   (vsip_length)fft_length, 1.0, VSIP_FFT_FWD, VSIP_ROW,
   0, 0);



/*********************************************************************
    Perform timing over N_ITS iterations.
 *********************************************************************/

/* Initialise the vsipl buffer.                                    */
set_matrix_data(buff_view, fft_length, no_of_ffts);

/* Start the timer...                                              */
startTime = get_cycles();

for(time_loop=0; time_loop<N_ITS; time_loop++)
{
   /* Carry out the fft with the vsipl version.                    */
   vsip_ccfftmip_f(buff_fft_plan, buff_view);

} /* End of timing loop.                                           */
/* End the timer...                                                */
ticks = (vsip_scalar_f)(get_cycles() - startTime);
```

```
      /* Output timings...                                                */
      printf("%g \n", (ticks/N_ITS) / clock_freq);

      /* Free up all views, blocks and plans.                             */
      vsip_fftm_destroy_f(buff_fft_plan);
      vsip_cmdestroy_f(buff_view);

   }

   return 1;

} /* End of time_multiple_fft.                                           */
```

9.3.5 Scalar and Vector Benchmark Code.

The following code segment shows how VSIPL blocks/views being created/initialised in order to benchmark the processing functions **vsip_svmul_f** and **vsip_vadd_f**.

```
/*************************************************************************/
/*                                                                       */
/*                    time_svmul_add()                                   */
/*                                                                       */
/* This benchmark function times the following operation                 */
/*      v1*s1+v2                                                         */
/* where v1 is vector 1, s1 is scalar 1 and v2 is vector 2. This is      */
/* out with the VSIPL calls vsip_svmul_f and vsip_vadd_f. The average    */
/* processing time is calculated and is output in microseconds.         */
/*                                                                       */
/* Processing lengths:                                                   */
/*     User given range of processing lengths.                           */
/*                                                                       */
/*************************************************************************/
vsip_scalar_i time_svmul_add(void)
{
   vsip_scalar_i index, test_no, v_length, time_loop;
   vsip_block_f *buff_block_inout  = NULL;
   vsip_vview_f *buff_view_inout   = NULL;

   vsip_block_f *buff_block_in_b   = NULL;
   vsip_vview_f *buff_view_in_b    = NULL;
   vsip_block_f *buff_block_tmp    = NULL;
   vsip_vview_f *buff_view_tmp     = NULL;
   vsip_scalar_f scalar_in_a       = 1.0;
   vsip_scalar_f clock_freq;
   vsip_scalar_f ticks;
   time_val startTime;



   /* Set the timer clock frequency.                                     */
   clock_freq = get_clock_frequency();


   /* Initialise the board and library.                                  */
   init_routine();
```

```
printf(" INPUT VECTOR  | OUTPUT VECTOR | TIMING.\n");

printf("-------------------------------------------\n");
for(test_no=0; test_no<N_TESTS; test_no++)
{
   /* Define test dimensions.                                          */
   v_length = test_buffer[test_no].v_length;

   printf(" %06d        | %06d        | ", v_length, v_length);


   /* Create data block and vector view for the VSIPL tests.           */
   buff_block_inout = vsip_blockcreate_f((size_t)(v_length), 0);
   if(buff_block_inout == NULL)
   {
      printf("Block creation error!\n");
      return 0;
   }
   buff_view_inout = vsip_vbind_f(buff_block_inout, 0, 1, v_length);
   if(buff_view_inout == NULL)
   {
      printf("View creation error!\n");
      return 0;
   }
   buff_block_in_b = vsip_blockcreate_f((size_t)(v_length), 0);
   if(buff_block_in_b == NULL)
   {
      printf("Block creation error!\n");
      return 0;
   }
   buff_view_in_b = vsip_vbind_f(buff_block_in_b, 0, 1, v_length);
   if(buff_view_in_b == NULL)
   {
      printf("View creation error!\n");
      return 0;
   }
   buff_block_tmp = vsip_blockcreate_f((size_t)(v_length), 0);
   if(buff_block_tmp == NULL)
   {
      printf("Block creation error!\n");
      return 0;
   }
   buff_view_tmp = vsip_vbind_f(buff_block_tmp, 0, 1, v_length);
   if(buff_view_tmp == NULL)
   {
```

```
      printf("View creation error!\n");
      return 0;
   }



   /*********************************************************************
       Perform timing over N_ITS iterations.
    *********************************************************************/

   /* Initialise the vsipl buffers.                                      */
   set_vector_data(buff_view_inout, v_length);
   set_vector_data(buff_view_in_b, v_length);
   scalar_in_a = 2.1;


   /* Start the timer...                                                 */
   startTime = get_cycles();

   for(time_loop=0; time_loop<N_ITS; time_loop++)
   {
      vsip_svmul_f(scalar_in_a,    buff_view_in_b, buff_view_tmp);
      vsip_vadd_f(buff_view_inout, buff_view_tmp,  buff_view_inout);

   } /* End of timing loop.                                              */

   /* End the timer...                                                   */
   ticks = (vsip_scalar_f)(get_cycles() - startTime);

   /* Output timings...                                                  */
   printf("%g \n", (ticks/N_ITS) / clock_freq);


   /* Free up all views, blocks and plans.                               */
   vsip_vdestroy_f(buff_view_inout);
   vsip_vdestroy_f(buff_view_in_b);
   vsip_vdestroy_f(buff_view_tmp);
}

   return 1;

} /* End of time_svmul_add                                              */
```

Chapter 10. Glossary

Admitted	*Block* state where the *data array* (memory) and associated *views* are available for VSIPL computations, and not available for user I/O or access.
Attribute	Characteristic or state of an object, such as *admitted* / *released*, *stride*, or *length*.
Binary Function	A function with two input arguments.
Block	A data storage abstraction representing contiguous data elements consisting of a *data array* and a VSIPL *block object*.
Block Object	Descriptor for a *data array* and its *attributes*, including a reference to the data array, the state of the block, data type and size.
Block Offset	The number of *elements* from the start of a *block*. A view with a block offset of zero starts at the beginning of the block.
Boolean	Used to represent the values of true and false, where false is always zero, and true is non-zero.
Bound	A *view* or *block* is bound to a *data array* if it references the data array.
Cloned View	An exact duplicate of a *view object*.
Column	Rightmost dimension in a *matrix*.
Column Stride	The number of *block elements* between successive elements within a *column*.
Complex Block	*Block* containing only complex *elements*. There are two formats for released complex blocks – *split* and *interleaved*. The complex data format for admitted complex blocks is not known to the user.
Conformant Views	*Views* that are the correct shape/size for a given computation.
const Object	An object that is not modified by the function, although data referenced by the const object may be modified.
Create	To allocate memory for an object and initialise it (if appropriate).
Data Array	Memory where data is stored.
Derived Block	A *real block* derived from a *complex block*. Note that the only way to create a derived block is to create a *derived view* of the real or complex component of a *split* complex view. In all other cases, retrieving the block from a view returns a reference to the original block.
Derived View	A derived view is a *view* created using a VSIPL function whose arguments include another view (a parent view). The derived view's data is some subset of the parent view's data. The data subset depends on the function call, and is physically co-located in memory with the parent view's data.
Destroy	To release the memory allocated to an object.
Development Library	An implementation of VSIPL that maximises error reporting at the possible expense of performance.

Domain	The set of all valid input values to a function.
Element	The atomic portion of data associated with a *block* or a *view*. For example, an element of a *complex block* of precision double is a complex number of precision double; for a view of type float an element is a single float number.
Hermitian Transpose	Conjugate transpose.
Hint	Information provided by the user to some VSIPL functions to aid optimization. Hints are optional and may be ignored by the implementation. Wrong hints may result in incorrect behavior.
In-Place	A type of algorithm implementation in which the memory used to hold the input to an algorithm is overwritten (completely or partially) with the output data. Often referred to in the context of an FFT algorithm.
Interleaved Complex	Storage format for *user data arrays* where the real and complex *element* components alternate in physical memory.
Kernel	The filter vector used in a FIR filter, or the vector or matrix used as the weights in a convolution.
Length	Number of *elements* in a view along a *view dimension*.
Matrix	A two-dimensional view.
Opaque	An opaque object may not be manipulated by simple assignment statements. Its attributes must be set/retrieved through access functions. All VSIPL objects are opaque.
Out-of-place	If none of the output views in a function call overlap the input views, the function is considered out-of-place.
Overlapped	Indicates that two or more *views* or *blocks* share one or more memory locations.
Production Library	A VSIPL implementation that maximises performance at the possible expense of not detecting user errors.
Range	Valid output values from a function.
Real Block	A *block* containing only real *elements*.
Region of Support	For neighborhood operations (i.e. FIR filtering, convolution), the non-zero values in the kernel, or the output. For example, a 3×3 FIR filter has a 'kernel region of support' of 3×3.
Released	*Block* state where the associated *data array* is available for user I/O and application access, but not available for VSIPL computations.
Row	Left-most dimension of a *matrix*.
Row Stride	The number of *block* elements between successive elements within a *row*.
Split Complex	Storage format for released *complex blocks* where the real *element* components are stored in one physically contiguous *data array*, and the imaginary components are stored in a separate physically contiguous data array.

Stride	Distance between successive *elements* of the block data array in a *view* along a view dimension. Strides can be positive, negative, or zero.
Subview	A *derived view* that describes a subset of the data from the original view, and is the same type as the original view.
Tensor	An *n*-dimensional matrix. VSIPL only supports three-dimensional tensors (3-tensor). The three dimensions are referred to as X, Y and Z.
Ternary Function	A function with three input arguments.
Unary Function	A function with a single input argument.
User Block	A block which is associated with user data arrays. User blocks are created in the released state and may be admitted and released.
User Data Array	Memory that has been allocated by the application for the storage of data using some functionality not part of the VSIPL standard.
Vector	A one-dimensional *view*.
View	A portion of a *block*, and a *view object* describing it. The view object has structural information allowing the data to be interpreted as a one-, two- or three-dimensional array for arithmetic processing.
View Dimension	A *view* represents a one-, two-, or three-dimensional data organisation termed respectively a *vector*, *matrix* or *tensor*. A *view dimension* represents one of the standard directions of these data representations.
View Object	A description of a portion of a *block*, including structural information that allows the data to be interpreted as a one-, two- or three-dimensional array for arithmetic processing. Attributes of the *view object* include *offset*, *stride*(s) and *length*(s).
VSIPL Block	*Block* referencing or *bound* to VSIPL data. A VSIPL block is created in the *admitted* state and may not be *released*.
VSIPL Data Array	Memory that has been allocated for the storage of data using some functionality that is part of the VSIPL standard.