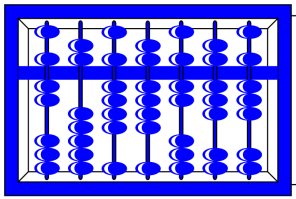


NASoftware Limited

Incorporating InfoSAR

VSIPL
Quick Reference Guide

VSIPL/Brief [3.20.14]

Release 3.20.14
February 2021

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.

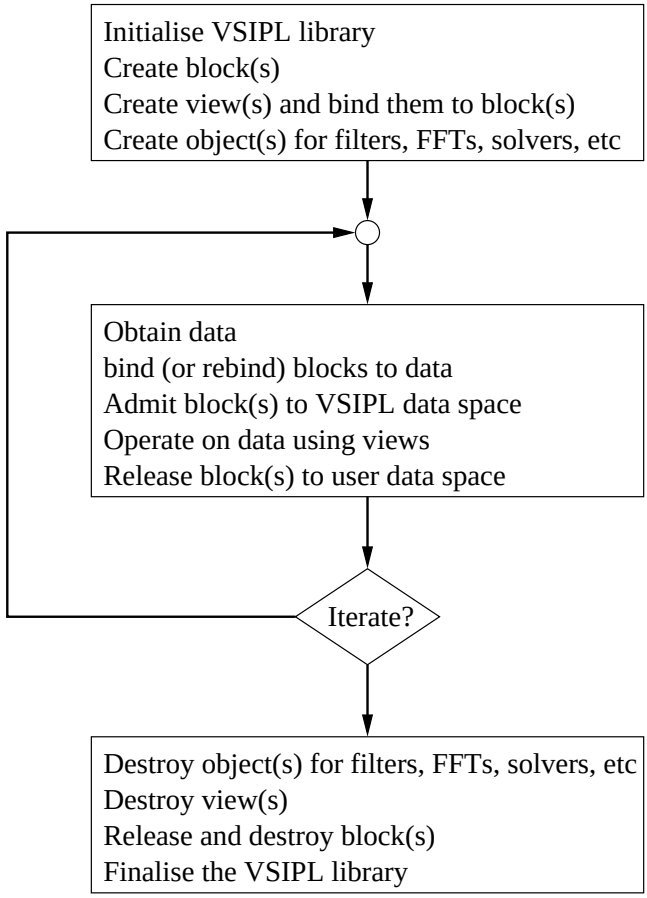
A block may have any number of views created on it; this allows the user to use

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

Prototype	Description
int vsip_init(void * ptr);	Provides initialization, allowing the library to allocate and set a global state, and prepare to support the use of VSIPL functionality by the user.
int vsip_finalize(void * ptr);	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.

3.2 Array and Block Functions

Prototype	Description
int vsip_Dblockadmit_P(vsip_Dblock_P * block, vsip_scalar_bl update);	Admit a data block for VSIPL operations. The following instances are supported: vsip_blockadmit_f vsip_blockadmit_i vsip_cblockadmit_f
*vsip_block_P ** vsip_blockbind_P(vsip_scalar_P * user_data, vsip_length num_items, vsip_memory_hint hint);	Create and bind a VSIPL block to a user-allocated data array. The following instances are supported: vsip_blockbind_f vsip_blockbind_i
*vsip_cblock_f ** vsip_cblockbind_f(vsip_scalar_f * user_data1, vsip_scalar_f * user_data2, vsip_length num_items, vsip_memory_hint hint);	Create and bind a VSIPL complex block to a user-allocated data array.
*vsip_Dblock_P ** vsip_Dblockcreate_P(vsip_length num_items, vsip_memory_hint hint);	Creates a VSIPL block and binds a (VSIPL-allocated) data array to it. The following instances are supported: vsip_blockcreate_f vsip_blockcreate_i vsip_cblockcreate_f
void vsip_Dblockdestroy_P(vsip_Dblock_P * block);	Destroy a VSIPL block object and any memory allocated for it by VSIPL. The following instances are supported: vsip_blockdestroy_f vsip_blockdestroy_i vsip_cblockdestroy_f

Prototype	Description
vsip_scalar_P * vsip_blockfind_P(const vsip_block_P * block);	Find the pointer to the data bound to a VSIPL released block object. The following instances are supported: vsip_blockfind_f vsip_blockfind_i
void vsip_cblockfind_f(const vsip_cblock_f * block, vsip_scalar_f ** user_data1, vsip_scalar_f ** user_data2);	Find the pointer(s) to the data bound to a VSIPL released complex block object.
vsip_scalar_P * vsip_blockrebind_P(vsip_block_P * block, vsip_scalar_P * new_data);	Rebind a VSIPL block to user-specified data. The following instances are supported: vsip_blockrebind_f vsip_blockrebind_i
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);	Rebind a VSIPL complex block to user-specified data.
vsip_scalar_P * vsip_blockrelease_P(vsip_block_P * block, vsip_scalar_bl update);	Release a VSIPL block for direct user access. The following instances are supported: vsip_blockrelease_f vsip_blockrelease_i
void vsip_cblockrelease_f(vsip_cblock_f * block, vsip_scalar_bl update, vsip_scalar_f ** user_data1, vsip_scalar_f ** user_data2);	Release a complex block from VSIPL for direct user access.
vsip_cmplx_mem vsip_cstorage(void);	Returns the preferred complex storage format for the system.

3.3 Vector View Functions

Prototype	Description
void vsip_Dvalldestroy_P(vsip_Dvview_P * vector);	Destroy a vector, its associated block, and any VSIPL data array bound to the block. The following instances are supported: vsip_valldestroy_f vsip_cvalldestroy_f
vsip_Dvview_P * vsip_Dvbind_P(vsip_Dblock_P * block, vsip_offset offset, vsip_stride stride, vsip_length length);	Create a vector view object and bind it to a block object. The following instances are supported: vsip_vbind_f vsip_vbind_i vsip_cvbind_f

Prototype	Description
vsip_Dvview_P * vsip_Dvcloneview_P(const vsip_Dvview_P * vector);	Create a clone of a vector view. The following instances are supported: vsip_vcloneview_f vsip_cvcloneview_f
vsip_Dvview_P * vsip_Dvcreate_P(vsip_length length, vsip_memory_hint hint);	Creates a block object and a vector view object of the block. The following instances are supported: vsip_vcreate_f vsip_cvcreate_f
vsip_Dblock_P * vsip_Dvdestroy_P(vsip_Dvview_P * vector);	Destroy a vector view object and return a pointer to the associated block object. The following instances are supported: vsip_vdestroy_f vsip_vdestroy_i vsip_cvdestroy_f
vsip_Dscalar_P vsip_Dvget_P(const vsip_Dvview_P * vector, vsip_index j);	Get the value of a specified element of a vector view object. The following instances are supported: vsip_vget_f vsip_cvget_f
void vsip_Dvgetattrib_P(const vsip_Dvview_P * vector, vsip_Dvattr_P * attr);	Return the attributes of a vector view object. The following instances are supported: vsip_vgetattrib_f vsip_vgetattrib_i vsip_cvgetattrib_f
vsip_Dblock_P * vsip_Dvgetblock_P(const vsip_Dvview_P * vector);	Get the block attribute of a vector view object. The following instances are supported: vsip_vgetblock_f vsip_cvgetblock_f
vsip_vview_f * vsip_vimagview_f(const vsip_cvview_f * complex_vector);	Create a vector view object of the imaginary part of a complex vector from a complex vector view object.
void vsip_Dvput_P(vsip_Dvview_P * vector, vsip_index j, vsip_Dscalar_P value);	Set the value of a specified element of a vector view object. The following instances are supported: vsip_vput_f vsip_cvput_f
vsip_Dvview_P * vsip_Dvputattrib_P(vsip_Dvview_P * ve3tor, const vsip_Dvattr_P * attr);	Set the attributes of a vector view object. The following instances are supported: vsip_vputattrib_f vsip_vputattrib_i vsip_cvputattrib_f

Prototype	Description

`vsip_Dvview_P *`
`vsip_Dvputlength_P(`
`vsip_Dvview_P * vector,`
`vsip_length length);`
	Set the length attribute of a vector view object.
	The following instances are supported:
	`vsip_vputlength_f`
	`vsip_cvputlength_f`

`vsip_Dvview_P *`
`vsip_Dvputoffset_P(`
`vsip_Dvview_P * vector,`
`vsip_offset offset);`
	Set the offset attribute of a vector view object.
	The following instances are supported:
	`vsip_vputoffset_f`
	`vsip_cvputoffset_f`

`vsip_Dvview_P *`
`vsip_Dvputstride_P(`
`vsip_Dvview_P * vector,`
`vsip_stride stride);`
	Set the stride attribute of a vector view object.
	The following instances are supported:
	`vsip_vputstride_f`
	`vsip_cvputstride_f`

`vsip_vview_f *`
`vsip_vrealview_f(`
`const vsip_cvview_f *`
`complex_vector);`
	Create a vector view object of the real part of a complex vector from a complex vector view object.

`vsip_Dvview_P *`
`vsip_Dvsubview_P(`
`const vsip_Dvview_P * vector,`
`vsip_index j,`
`vsip_length n);`
	Create a vector view object that is a subview of a vector view object.
	The following instances are supported:
	`vsip_vsubview_f`
	`vsip_cvsubview_f`

Chapter 4. Scalar Functions

4.1 Complex Scalar Functions

Prototype	Description
void vsip_CMPLX_f(vsip_scalar_f a, vsip_scalar_f b, vsip_cscalar_f * r);	Form a complex scalar from two real scalars.
vsip_cscalar_f vsip_cmplx_f(vsip_scalar_f re, vsip_scalar_f im);	Form a complex scalar from two real scalars.
vsip_scalar_f vsip_imag_f(vsip_cscalar_f x);	Extract the imaginary part of a complex scalar.
vsip_scalar_f vsip_real_f(vsip_cscalar_f x);	Extract the real part of a complex scalar.

Chapter 5. Random Number Generation

5.1 Random Number Functions

Prototype	Description

`vsip_randstate *`
`vsip_randcreate(`
`const vsip_index seed,`
`const vsip_index numprocs,`
`const vsip_index id,`
`const vsip_rng portable);`
	Create a random number generator state object.

`int`
`vsip_randdestroy(`
`vsip_randstate * rand);`
	Destroys (frees the memory used by) a random number generator state object. Returns zero on success, non-zero on failure.

`void`
`vsip_vrandu_f(`
`vsip_randstate * state,`
`const vsip_vview_f * R);`
	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 $(0, 2^{31} - 1)$.

Chapter 6. Elementwise Functions

6.1 Elementary Mathematical Functions

Prototype	Description
void vsip_vatan_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the principal radian value in $[-\pi/2, \pi/2]$ of the inverse tangent for each element of a vector.
void vsip_vatan2_f(const vsip_vview_f * A, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the four-quadrant radian value in $[-\pi, \pi]$ of the inverse tangent of the ratio of the elements of two input vectors.
void vsip_vcos_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the cosine for each element of a vector. Element angle values are in radians.
void vsip_vexp_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the exponential function value for each element of a vector.
void vsip_vlog_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the natural logarithm for each element of a vector.
void vsip_vlog10_f(const vsip_vview_f * A, const vsip_vview_f * R);	Compute the base ten logarithm for each element of a vector.
void vsip_vsin_f(const vsip_vview_f * A, const vsip_vview_f * R);	Compute the sine for each element of a vector. Element angle values are in radians.
void vsip_vsqrt_f(const vsip_vview_f * A, const vsip_vview_f * R);	Compute the square root for each element of a vector.

6.2 Unary Operations

Prototype	Description
void vsip_cvconj_f(const vsip_cvview_f * A, const vsip_cvview_f * R);	Compute the conjugate for each element of a complex vector.
void vsip_Dvmag_P(const vsip_Dvview_P * A, const vsip_vview_P * R);	Compute the magnitude for each element of a vector. The following instances are supported: vsip_vmag_f vsip_cvmag_f
void vsip_vcmagsq_f(const vsip_cvview_f * A, const vsip_vview_f * R);	Computes the square of the magnitudes for each element of a vector.

Prototype	Description
void vsip_Dvneg_P(const vsip_Dvview_P * A, const vsip_Dvview_P * R);	Computes the negation for each element of a vector. The following instances are supported: vsip_vneg_f vsip_cvneg_f
void vsip_vrecip_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the reciprocal for each element of a vector.
void vsip_vsq_f(const vsip_vview_f * A, const vsip_vview_f * R);	Computes the square for each element of a vector.
vsip_scalar_f vsip_vsumval_f(const vsip_vview_f * A);	Returns the sum of the elements of a vector.
vsip_scalar_f vsip_vsumsqval_f(const vsip_vview_f * A);	Returns the sum of the squares of the elements of a vector.

6.3 Binary Operations

Prototype	Description
void vsip_Dvadd_P(const vsip_Dvview_P * A, const vsip_Dvview_P * B, const vsip_Dvview_P * R);	Computes the sum, by element, of two vectors. The following instances are supported: vsip_vadd_f vsip_cvadd_f
void vsip_svadd_f(const vsip_scalar_f a, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the sum, by element, of a scalar and a vector.
void vsip_vdiv_f(const vsip_vview_f * A, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the quotient, by element, of two vectors.
void vsip_svdiv_f(const vsip_scalar_f a, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the quotient, by element, of a scalar and a vector.
void vsip_cvjmul_f(const vsip_cvview_f * A, const vsip_cvview_f * B, const vsip_cvview_f * R);	Computes the product of a complex vector with the conjugate of a second complex vector, by element.
void vsip_Dvmul_P(const vsip_Dvview_P * A, const vsip_Dvview_P * B, const vsip_Dvview_P * R);	Computes the product, by element, of two vectors. The following instances are supported: vsip_vmul_f vsip_cvmul_f

Prototype	Description
void vsip_rcvmul_f(const vsip_vview_f * A, const vsip_cvview_f * B, const vsip_cvview_f * R);	Computes the product, by element, of two vectors.
void vsip_Dsvmul_P(const vsip_Dscalar_P a, const vsip_Dvview_P * B, const vsip_Dvview_P * R);	Computes the product, by element, of a scalar and a vector. The following instances are supported: vsip_svmul_f vsip_csvmul_f
void vsip_rscvmul_f(const vsip_scalar_f a, const vsip_cvview_f * B, const vsip_cvview_f * R);	Computes the product, by element, of a real scalar and a complex vector.
void vsip_Dvsub_P(const vsip_Dvview_P * A, const vsip_Dvview_P * B, const vsip_Dvview_P * R);	Computes the difference, by element, of two vectors. The following instances are supported: vsip_vsub_f vsip_cvsub_f

6.4 Selection Operations

Prototype	Description
void vsip_vmax_f(const vsip_vview_f * A, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the maximum, by element, of two vectors.
vsip_scalar_f vsip_vmaxval_f(const vsip_vview_f * A, vsip_index * index);	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.
void vsip_vmin_f(const vsip_vview_f * A, const vsip_vview_f * B, const vsip_vview_f * R);	Computes the minimum, by element, of two vectors.
vsip_scalar_f vsip_vminval_f(const vsip_vview_f * A, vsip_index * index);	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.

6.5 Element Generation and Copy

Prototype	Description
void vsip_Dvcopy_P_P(const vsip_Dvview_P * A, const vsip_Dvview_P * R);	Copy the source vector to the destination vector performing any necessary type conversion of the standard ANSI C scalar types. The following instances are supported: vsip_vcopy_f_f vsip_vcopy_f_i vsip_vcopy_i_f vsip_cvcopy_f_f
void vsip_vfill_f(const vsip_scalar_f a, const vsip_vview_f * R);	Fill a vector with a constant value.
void vsip_vramp_f(const vsip_scalar_f alpha, const vsip_scalar_f beta, const vsip_vview_f * R);	Computes a vector ramp by starting at an initial value and incrementing each successive element by the ramp step size.

6.6 Manipulation Operations

Prototype	Description
void vsip_vcmplx_f(const vsip_vview_f * A, const vsip_vview_f * B, const vsip_cvview_f * R);	Form a complex vector from two real vectors.
void vsip_vimag_f(const vsip_cvview_f * A, const vsip_vview_f * R);	Extract the imaginary part of a complex vector.
void vsip_vreal_f(const vsip_cvview_f * A, const vsip_vview_f * R);	Extract the real part of a complex vector.

Chapter 7. Signal Processing Functions

7.1 FFT Functions

Prototype	Description
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);	Create a 1D FFT object.
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);	Create a 1D FFT object.
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);	Create a 1D FFT object.
int vsip_fft_destroy_f(vsip_fft_f * plan);	Destroy an FFT object.
void vsip_ccfftop_f(const vsip_fft_f * plan, const vsip_cvview_f * x, const vsip_cvview_f * y);	Apply a complex-to-complex Fast Fourier Transform (FFT).
void vsip_crfftop_f(const vsip_fft_f * plan, const vsip_cvview_f * x, const vsip_vview_f * y);	Apply a complex-to-real Fast Fourier Transform (FFT).
void vsip_rcfftop_f(const vsip_fft_f * plan, const vsip_vview_f * x, const vsip_cvview_f * y);	Apply a real-to-complex Fast Fourier Transform (FFT).

7.2 Filter Functions

Prototype	Description
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);	Create a decimated FIR filter object. The following instances are supported: vsip_fir_create_f vsip_cfir_create_f

Prototype	Description

`int`
`vsip_Dfir_destroy_P(`
`vsip_Dfir_P * plan);`
	Destroy a FIR filter object.
	The following instances are supported:
	`vsip_fir_destroy_f`
	`vsip_cfir_destroy_f`

`int`
`vsip_Dfirflt_P(`
`vsip_Dfir_P * plan,`
`const vsip_Dvview_P * x,`
`const vsip_Dvview_P * y);`
	FIR filter an input sequence and decimate the output.
	The following instances are supported:
	`vsip_firflt_f`
	`vsip_cfirflt_f`

7.3 Miscellaneous Signal Processing Functions

Prototype	Description

`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);`
	Compute the histogram of a vector.

Chapter 8. Linear Algebra

8.1 Matrix and Vector Operations

Prototype	Description
vsip_cscalar_f vsip_cvjdot_f(const vsip_cvview_f * A, const vsip_cvview_f * B);	Compute the conjugate inner (dot) product of two complex vectors.
vsip_Dscalar_P vsip_Dvdot_P(const vsip_Dvview_P * A, const vsip_Dvview_P * B);	Compute the inner (dot) product of two vectors. The following instances are supported: vsip_vdot_f vsip_cvdot_f

Chapter 9. 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.