TASP VSIPL Core

Randall Judd

This document is a description of, and a manual for, an implementation of the VSIPL Core Profile by the Tactical Advanced Signal Processing Common Operating Environmnet working group. This work is supported by PEO (USW) PMS411, Tony Lee.

Space and Naval Warfare Systems Center
San Diego
D881

TABLE OF CONTENTS

REQUIRED CORE PUBLIC TYPES

CORE FUNCTION LIST

CORE FUNCTION LIST

CHAPTER 1 Introduction To TASP VSIPL and the Core Profile

Introduction

This book describes the functionality of the Core Profile defined for the Vector/Signal/Image processing (VSIP) Library (VSIPL) as developed by the TASP (Tactical Advanced Signal Processing) COE (Common Operating Environment) effort.

This book is not a copy of, nor a replacement for, the VSIPL specification.

Code History

The original code basis for the library was a pre-alpha (incomplete) version of the VSIPL Reference library produced by Hughes Research Laboratory of Malibu, California in December of 1997. This library has been greatly reorganized and modified to fit a format more suitable to the author’s vision of a VSIP library, and also the author’s programing ability. The original December release was template based using m4 as a code generator. The author’s method was to copy the generated C files and header files and modify them directly, instead of trying to maintain a template method he did not understand. In addition many changes have been made to the library to add performance, and to keep up with the changing VSIPL specification.

TASP and the TASP COE

The TASP group started out as an effort by NAVSEA PMS 428 (Now PMS 411) to do a procurement of COTS signal processing hardware for DOD use similar to the TAC program. One of the goals of the TASP group was to foster a Common Operating Environment (COE) for signal processing. Without a COE for signal processing the software upkeep cost of COTS signal processing hardware will be prohibitive.

In the last few years the methods used by DOD to procure hardware have changed, and eventually the TASP effort for hardware procurement was abandoned. However the COE effort is still important and has survived. One of the elements of a COE for Signal Processing is a common signal processing library supported by multiple vendors. The TASP group has decided to support the VSIP Library Forum effort to produce a de facto signal processing standard, and eventually an actual standard, for a signal processing library. If successful VSIPL will be used for the TASP COE signal processing library.

The VSIP Library Effort and the VSIPL Forum

The VSIP library (VSIPL) effort was initially funded by DARPA and headed by Hughes Research Lab (HRL) (David Schwartz). HRL has changed their name and are now called HRL Laboratories, LLC. HRL no longer stands for Hughes Research Lab.

The main goal of the VSIPL Forum is to produce a signal processing library specification suitable for a wide variety of embedded hardware. The specification will allow vendors to write an efficient and fast library implementation for their product, and at the same time will allow VSIPL application programers to write portable code which will run on a variety of VSIPL compliant hardware without major porting efforts.

Many groups have participated in the VSIPL Forum. For a more complete list of participants one should refer to the VSIPL specification Acknowledgment section.

The primary external funding for the forum was from DARPA, and from TASP; however most companies participated with no external funding.

The TASP VSIPL Demonstration Library

It should be made clear that this document is a product of the TASP effort, and the associated library is also a TASP effort. These products are considered to be separate from the VSIPL effort. We are not trying to do a separate specification, of course, but just as vendors will write their own VSIPL compliant library for their product as independent agents, without the desire for, or need of, the VSIPL Forum telling them how to go about their business, so too this effort is independent of the VSIPL forum.

Since the author is an active participant of the VSIPL Forum people may get the idea that this document, or the associated library effort have somehow been blessed as part of the specification. This is not the case. The author felt the need for a certain amount of independence in developing this library in order to produce a product for demonstration purposes at the earliest time. As such he felt it was necessary to make design decisions independent of any other agent.

For this reason people should be cautious and view this library as what it is. The advantage of being independent is the ability to get a lot of work done. The disadvantage is you may screw up the implementation, and depart from the specification. This implementation is not well tested. It attempts to be VSIPL compliant, but it may have some problems. Many functions were completed with an eye toward getting something on the road versus writing a really good function. Participants who find departures from the VSIPL specification within this document, or within the library, or who find software or algorithm errors, are encouraged to contact the author via email (judd@spawar.navy.mil).

As time goes by, if VSIPL is successful, the TASP VSIPL implementation may eventually become very good; or if another better public domain VSIPL implementation becomes available, this library may become unused. In any case view the results of every function with a certain amount of caution. No claims are made that the associated library, or this document, are good for any purpose.

The Core Profile

The entire function list defined by the VSIPL specification is very large, and it would be prohibitively expensive to produce optimized code on embedded hardware for the entire library. Several of the signal processing hardware and software vendors proposed a profile of the library that was very small and was, they felt, usable and relatively inexpensive to produce. This profile, called Core Lite, has only 126 required functions. A somewhat larger profile which includes 511 functions was also defined and has been called Core. This book covers the TASP VSIPL implementation of the Core Profile. For readers who have read the Core Lite Book the first two chapters of this book are very similar, except that chapter two will cover many more function prototypes.

VSIP Fundamentals

A little background into the VSIP methods used for defining data types and functions is needed before the function list and programing methods are introduced. Some of this information is specific to the method used within the TASP VSIPL implementation, and it is not necessarily true that other implementation would use the same method. The VSIPL specification tries to abstract the method of achieving the end away from the end itself. As long as a vendor uses the proper API and meets all the rules, the exact method for achieving the correct result is not of concern to VSIPL. However talking in abstract terms sometimes leads to confusion, so the author will be a little more direct in talking about how the TASP VSIPL implementation achieved the desired result. This should make it easier for the user to understand the implementation and how to use VSIPL. After one library is learned, any other compliant libraries used will be the same, no matter what internal (private) methods were used to define the VSIPL objects.

One should not bring excess programing definitions to this document. VSIPL is an object based method, but it is not object oriented in the strict sense. The author is not knowledgeable of object oriented terminology or programing. The author knows how the forum uses certain terms, and that there are regular discussions (or arguments) about using some terms improperly. For the purpose of this document the author will attempt to explain how he is using the term, and hopefully the reader will not be to critical.

VSIPL Objects and Data Types

Roughly speaking VSIPL has three basic types.

The first base type is a VSIPL *scalar.* Frequently these are just typedefs of ANSI C types to a VSIPL naming convention. For instance vsip_scalar_f is an ANSI C float (float), and vsip_scalar_i is an ANSI C integer (int). The author feels it is important that people use the VSIPL types in their programing. Because every function within VSIPL is strongly typed this will keep help keep you honest and will reduce errors. In addition if the need arises to port some code from say a float library to a double library it simplifies the porting. This will become more obvious as you learn more about the library. In VSIPL Core there are scalars of type vector index, matrix index, boolean, float, integer, and complex float.

The next base type is a *block.* A block is equivalent to a memory storage area of a particular data type and some size. The data is stored in sequential element locations, as far as the application programer is concerned. In TASP VSIPL a block object is actually an abstract data type (ADT) with variables to hold the block length, information about the block's state (more about this later) and a pointer to some physical memory. For complex blocks (in TASP VSIPL) there are actually two pointers to two real blocks, and information concerning the data layout of those blocks. All you need to know about blocks to program portable code is that from the point of view of the VSIPL functions that work on blocks they are a chunk of sequential VSIPL elements of a particular type, the first element location being at 0 (zero) and the last element being at $N-1$ where N is the size of the block. In TASP VSIPL Core there are blocks of type float, complex float, integer, vector index, matrix index, and boolean.

The next, and final basic type, is a *view*. In TASP VSIPL Core, there are only views of type vector and matrix float, vector and matrix complex float, vector integer, vector vector index, vector matrix index, and vector boolean. The view, similar to the block, is an abstract data type. The view holds all the information needed to access some particular portion of a blocks data. For the TASP VSIPL implementation the view has a block pointer which is set equal to the block whose data it references. All data is referenced through this block pointer. In addition the view holds an offset from the beginning of the block (starting at zero), a length or lengths (of the vector or matrix) and a stride or strides (along the view dimensions) through the block. The stride indicates the distance between consecutive view elements within the block for a particular dimension. The stride, along with the offset is used in conjunction with the vector or matrix indices to map the vector or matrix onto the block. A stride of 1 is every element, a stride of -1 is also every element, but the view goes through the block in the opposite direction. A stride of zero will select a particular element as a constant vector at the offset location.

In order to produce portable VSIPL application code the application programer must use only VSIPL function calls to use or modify *blocks* or *views*. To enforce this all the abstract data types used by VSIPL for its internal workings are created as incomplete data types. Because of this, unless the private header files where the blocks and view data types are completed are available, the user must use VSIPL function calls. The use of abstract data types and incomplete type definitions leads one to call *views* and *blocks* objects.

For more information on VSIPL design requirements see Appendix A, or obtain the VSIPL specification (when it is completed).

A Simple First Example

Except for the list of functions in Chapter 2, and Appendices which may cover any topic, the rest of this document will be done in a tutorial fashion. In general the method used will be to produce a simple example with an exhaustive explanation. Within the explanation many important principles for successful VSIPL programs will be explained. All examples in this document will be limited to code that will compile on VSIPL libraries that conform to the Core Profile

Add two vectors example.

If this were being done in Matlab this example would look as follows:

```
>> A=[0:7]
A =
     0     1     2     3     4     5     6     7
>> B(A+1)=5
B =
     5     5     5     5     5     5     5     5
>> C=A+B
C =
     5     6     7     8     9    10    11    12
```

Now lets do this in VSIPL Code:

Example 1

```
#include<stdio.h>
#include<vsip.h>

#define N 8 /* the length of the vector */

int main()
{
  void VU_vprint_f(vsip_vview_f*);
  vsip_vview_f   *A = vsip_vcreate_f(N,0),
                 *B = vsip_vcreate_f(N,0),
                 *C = vsip_vcreate_f(N,0);
  vsip_vramp_f(0,1,A);
  printf("A = \n");VU_vprint_f(A);

  vsip_vfill_f(5,B);
  printf("B = \n");VU_vprint_f(B);

  vsip_vadd_f(A,B,C);
  printf("C = \n");VU_vprint_f(C);

  vsip_valldestroy_f(A);
  vsip_valldestroy_f(B);
  vsip_valldestroy_f(C);
  return 1;
}

void VU_vprint_f(vsip_vview_f* a){
  vsip_length i;
  for(i=0; i<vsip_vgetlength_f(a); i++)
  printf("%4.0f",vsip_vget_f(a,i));
  printf("\n");
  return;
}
```

The above program produces the following output:

```
A =
   0   1   2   3   4   5   6   7
B =
   5   5   5   5   5   5   5   5
C =
   5   6   7   8   9  10  11  12
```

Let’s examine Example 1.

On **line 2** we include the **`vsip.h`** header file. This will be needed in every program using VSIPL code, and a compliant library is required to provide a header file called vsip.h.

On **line 9** we have our first VSIPL function call. Note that the VSIPL vector view objects A, B and C are type defined to a pointer of type **`vsip_vview_f`** and then assigned a value by the function **`vsip_vcreate_f`**.

The **`create`** function is a convenience function, subsuming the block create and view bind jobs into one function. The **`vsip_vcreate_f`** function will create a block of type real float (with some state we talk about in chapter 3), create a data space of sufficient size to hold *N* real float values and then attaches this to the block, and creates a vector view of type real float and binds the block to this view. The vector view (or just vector) is created with a length of **`N`** elements, an offset of zero, and a stride of one, so that the vector is of an exact size to view the entire block.

An important item to note here is that VSIPL has allocated space in memory for three items. These are the space for the block object (block ADT), space for the data storage (the data array) and space for the vector object (vector ADT). All of this memory must be destroyed, when no longer needed, to prevent memory leaks.

Blocks and associated data arrays created by the VSIPL create functions are always created and destroyed together. Whenever you do a block create and a block destroy the actions to create the block, and its associated data, or destroy a block and its associated data happen together. Generally we will only say we create a block of length N, or destroy a block.

We have also created a vector view. There may be many vector views associated with a block. There are, of course, functions to destroy a vector view. These will generally not destroy the block also. It is important that the application programer keep track of what views are attached to a block and only destroy the block after all the views binding the block have been destroyed.

Note that all VSIPL functions which allocate memory return a null pointer if the memory allocation fails. We have not checked for an allocation failure in our example, but the check is recommended.

In our example above on **lines 21-23** we see where the object destruction takes place. We know, since this program is short and we kept track, that each block is bound by exactly one view. To destroy our objects we use a convenience function which will destroy the view, the block, and any VSIPL allocated data array associated with the block.

The other important items in our code reside on **lines 12**, **15**, and **18**. These are self explanatory as to function, but note that no stride, length or offset information are included in the arguments to these functions. All this information resides in the vector. So, for instance, the ramp function has a starting value of zero, an increment of one, but no stopping point. The function just goes until the vector is full. Since we set the vector length to eight, we get a vector running from zero to seven.

In order to see our output we wrote a vector print function. For VSIPL user functions the author uses a prefix of VU_ for VSIPL User. The print function starts at **line 27**.

We note that internal to the VSIPL library the author uses the prifix VI_ for VSIPL Implementor. **No VI_ function should be used in any user code**; however if your writing your own library, or modifying the author's, feel free.

List of Acronyms

1.	TASP	Tactical Advanced Signal Processing
2.	COE	Common Operating Environment
3.	VSIPL	Vector/Signal/Image Processing Library
4.	COTS	Commercial Of the Shelf
5.	TAC	Tactical Advanced Computer
6.	API	Application Program Interface
7.	ADT	Abstract Data Type
8.	DOD	Department of Defense

CHAPTER 2 Core Functions

Introduction

This section includes an alphabetical listing of all VSIPL functionality included with a minimal VISIPL Core Library, and a listing of all public type definitions (enumerated types and structures available in the header file) needed by the core functions. Each function is listed with a functionality statement, the function prototype, and a description of each function argument. The author recommends only browsing this chapter lightly. Its purpose is as a reference. In the VSIPL specification many of the functions names have been generalized to include all precisions. The names in this document are not generalized and reflect the TASP VSIPL Core implementation of a minimal VSIPL Core profile encompassing ANSI C float and int.

The TASP VSIPL core distribution includes additional functionality other than that required by a minimal core distribution. Proper prototypes for much of the additional functionality may be derived from the listed functionality by replacing the precision argument with the required precision, for instance the _f goes to an _d when deriving the double function prototype from the float function prototype.

In order to have some reasonable ordering of the functions the alphabetical listing is based upon a root function name, not the actual vsip function. For instance the second function in the list is the “add” function. There are several add functions in the Core profile. All of them are placed together under add.

When a function requires a special object it needs support functions to create the object, and destroy it, and perhaps query it for its attributes. For instance to do a discrete fourier transform one needs a function to create an FFT object, a function to do the actual FFT using the FFT object, and a function to destroy the FFT object when it is no longer needed. The author calls functions which are designed to work together to do a single job *function sets*. Function sets are placed together under a single heading. For instance all the functions involved with doing an FFT are placed under the FFT heading.

In addition the Ternary functions included in the core profile (functions requiring three inputs) are listed in a ternary functions section. The root names for ternary functions are not very descriptive.

No attempt is made to be exhaustive in the function descriptions. Those interested in more detail are directed to the VSIPL specification document available on the internet site. (www.vsipl.org) In addition various examples included in this document will provide more detail on the use of some of the more complicated functions.

Required Core Public Types

This section covers the enumerated types and special structures needed by the core functions. These are defined in the public header file **`vsip.h`**.

`alg_hint`

```
typedef enum {
   VSIP_ALG_TIME = 0,
   VSIP_ALG_SPACE = 1,
   VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Algorithm hint used in create functions to indicate to the implementation how the user would like the created object to be used. ALG_TIME would indicate a desire for the fastest result, ALG_SPACE for the least memory usage, and ALG_NOISE for the most accurate. Not required to be supported and not supported in TASP VSIPL. Any valid hint may be used.

```
Function List where used
   vsip_conv1d_create_f
   vsip_corr1d_create_f
   vsip_ccorr1d_create_f
   vsip_ccfftop_create_f
   vsip_ccfftip_create_f
   vsip_rcfftop_create_f
   vsip_crfftop_create_f
   vsip_ccfftmop_create_f
   vsip_ccfftmip_create_f
   vsip_rcfftmop_create_f
   vsip_crfftmop_create_f
   vsip_fir_create_f
   vsip_cfir_create_f
```

`bias`

```
typedef enum{
   VSIP_BIASED = 0,
   VSIP_UNBIASED = 1
} vsip_bias;
```

Flag to indicate whether a biased or unbiased result is desired.

```
Function List where used
   vsip_correlate1d_f
   vsip_ccorrelate1d_f
```

`chol_attr`

```
typedef struct{
   vsip_mat_uplo uplo;
   vsip_length n;
} vsip_chol_attr_f;
```

```
typedef struct{
   vsip_mat_uplo uplo;
   vsip_length n;
} vsip_cchol_attr_f;
```

Attributes structure for the Cholesky decomposition object. Used with the Cholesky get attributes function.

cmplx_mem

```
typedef enum {
   VSIP_CMPLX_INTERLEAVED,
   VSIP_CMPLX_SPLIT,
   VSIP_CMPLX_NONE
} vsip_cmplx_mem
```

Used to indicate the type of user complex data array is optimal for the implementation. The NONE type indicates either interleaved or split work equally well. Used as a return value for **`vsip_cstorage`**.

conv1d_attr

```
typedef struct {
   vsip_length kernel_len;
   vsip_symmetry symm;
   vsip_length data_len;
   vsip_support_region support;
   vsip_length out_len;
   vsip_length decimation;
} vsip_conv1d_attr_f;
```

Attributes structure for the convolution object. Used with the convolution get attributes function.

corr1d_attr

```
typedef struct {
   vsip_length ref_len;
   vsip_length data_len;
   vsip_support_region support;
   vsip_length lag_len;
} vsip_corr1d_attr_f;
```

```
typedef struct {
   vsip_length ref_len;
   vsip_length data_len;
   vsip_support_region support;
   vsip_length lag_len;
} vsip_ccorr1d_attr_f;
```

Public attributes structure for the correlation object. Used with the correlation get attributes function.

fir_attr

```
typedef struct {
   vsip_scalar_vi kernel_len;
```

```
      vsip_symmetry symm;
      vsip_scalar_vi in_len;
      vsip_scalar_vi out_len;
      vsip_length decimation;
      vsip_obj_state state;
   } vsip_fir_attr_f;
   typedef struct {
      vsip_scalar_vi kernel_len;
      vsip_symmetry symm;
      vsip_scalar_vi in_len;
      vsip_scalar_vi out_len;
      vsip_length decimation;
   } vsip_cfir_attr_f;
```

Public attributes structure for the FIR object. Used with the FIR get attributes function.

fft_attr

```
   typedef struct {
      vsip_scalar_vi input;
      vsip_scalar_vi output;
      vsip_fft_place place;
      vsip_scalar_f scale;
      vsip_fft_dir dir;
   } vsip_fft_attr_f;
```

Public attributes structure for the FFT object. Used with the FFT get attributes function.

fft_dir

```
   typedef enum {
      VSIP_FFT_FWD = -1,
      VSIP_FFT_INV = 1
   } vsip_fft_dir;
```

Direction argument for the fft create functions used to indicate the direction of the FFT.

Function List where used
```
      vsip_ccfftop_create_f
      vsip_ccfftip_create_f
      vsip_ccfftmop_create_f
      vsip_ccfftmip_create_f
```

fft_place

```
   typedef enum {
      VSIP_FFT_IP = 0,
      VSIP_FFT_OP = 1
   } vsip_fft_place;
```

fftm_attr

```
   typedef struct {
      vsip_scalar_vi input;
      vsip_scalar_vi output;
      vsip_fft_place place;
      vsip_scalar_f scale;
      vsip_fft_dir dir;
      vsip_major major;
   } vsip_fftm_attr_f;
```

Public attributes structure for the multiple FFT object. Used with the multiple FFT get attributes function.

lu_attr

```
typedef struct {
   vsip_length n;
} vsip_lu_attr_f;
```

```
typedef struct {
   vsip_length n;
} vsip_clu_attr_f;
```

major

```
typedef enum{
   VSIP_ROW = 0,
   VSIP_COL = 1,
}vsip_major;
```

mat_op

```
typedef enum {
   VSIP_MAT_NTRANS = 0,
   VSIP_MAT_TRANS = 1,
   VSIP_MAT_HERM = 2,
   VSIP_MAT_CONJ = 3
}vsip_mat_op;
```

mat_side

```
typedef enum{
   VSIP_MAT_LSIDE = 0,
   VSIP_MAT_RSIDE =1
} vsip_mat_side;
```

mattr

```
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
   vsip_stride col_stride;
   vsip_length col_length;
   vsip_block_f* block;
} vsip_mattr_f;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
```

```
      vsip_stride col_stride;
      vsip_length col_length;
      vsip_cblock_f* block;
   } vsip_cmattr_f;
```

Public matrix attributes. Used by matrix get attributes to retrieve the attributes of a matrix view and by put matrix attributes to set the attributes of a matrix. The block attribute of a view may not be set, except at creation, and is ignored on a put.

memory_hint

```
   typedef enum {
      VSIP_MEM_NONE = 0,
      VSIP_MEM_RDONLY = 1,
      VSIP_MEM_CONST = 2,
      VSIP_MEM_SHARED = 3,
      VSIP_MEM_SHARED_RDONLY = 4,
      VSIP_MEM_SHARED_CONST = 5
   }vsip_memory_hint;
```

Enumerated typedef indicating what type of memory the user would like allocated by VSIPL. The TASP VSIPL implementation of core does not use this memory hint for anything. Note the use of the overloaded depth (*d*), shape (*s*), and precision (*p*) below in the function list where used.

```
   Function List where used
      vsip_vcreate_blackman_f
      vsip_vcreate_kaiser_f
      vsip_vcreate_hanning_f
      vsip_vcreate_cheyby_f
      vsip_dblockcreate_p
      vsip_dsviewcreate_p
      vsip_dsblockbind_p
```

obj_state

Enumerated type indicating if an object which saves state information between calls should save the state, or act as if it were freshly created at each call. Currently only used for the FIR function set.

```
   typedef enum {
      VSIP_STATE_NO_SAVE = 1,
      VSIP_STATE_SAVE = 2
      } vsip_obj_state;
```

qrd_attr

Public QRD attribute object.

```
   typedef struct {
      vsip_length m;
      vsip_length n;
      vsip_qrd_opt Qopt;
   } vsip_qrd_attr_f;
```

```
typedef struct {
   vsip_length m;
   vsip_length n;
   vsip_qrd_opt Qopt;
} vsip_cqrd_attr_f;
```

qrd_prob

```
typedef enum{
   VSIP_QRD_COV
   VSIP_QRD_LLSQ
} vsip_qrd_prob
```

qrd_qopt

Enumerated typedef indicating what type of QRD information is saved in the QRD object. For an m by n where matrix $\bar{A} = \bar{Q}\bar{R}$ then for option **NOSAVEQ** only the $\bar{R}$ information is saved, for option **SAVEQ** the entire $\bar{Q}$ is saved, and for option **SAVEQ1** only the skinny $\overline{Q_1}$ which encompasses the range of $\bar{A}$ is saved. Note although the author has said that the $\bar{Q}$ is saved what is actually saved in the QRD object is vendor dependent. Only the information necessary to do the calculations defined on the QRD object need be saved. How this is done is vendor dependent.

```
typedef enum{
   VSIP_QRD_NOSAVEQ = 0,
   VSIP_QRD_SAVEQ = 1,
   VSIP_QRD_SAVEQ1 = 2
} vsip_qrd_qopt
```

support_region

```
typedef enum {
   VSIP_SUPPORT_FULL = 0,
   VSIP_SUPPORT_SAME = 1,
   VSIP_SUPPORT_MIN = 2
} vsip_support_region;
```

symmetry

```
typedef enum {
   VSIP_NONSYM = 0,
   VSIP_SYM_EVEN_LEN_ODD = 1,
   VSIP_SYM_EVEN_LEN_EVEN = 2
} vsip_symmetry;
```

vattr

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_f* block;
} vsip_vattr_f;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_cblock_f* block;
} vsip_cvattr_f;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_i* block;
} vsip_vattr_i;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_vi* block;
} vsip_vattr_vi;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_mi* block;
} vsip_vattr_mi;
```

```
typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_bl* block;
} vsip_vattr_bl;
```

Public vector attributes. Used by vector get attributes to retrieve the attributes of a vector and by put vector attributes to set the attributes of a vector. The block attribute may not be set in a view, except on view create, and is ignored on a put.

rng

```
typedef enum {
   VSIP_PRNG = 0,
   VSIP_NPRNG = 1
   } vsip_rng;
```

Indicates to the random create function whether an implementation dependent non-portable random number generator (NPRNG), or the portable random number generator defined by the VSIPL specification (PRNG) is desired.

Core Function List

acos

Inverse Cosine function.

```
void vsip_acos_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2);
```

Argument a1	Input vector.
Argument a2	Output vector.

add

Add two scalars.

```
void vsip_CADD_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f* a3);
```

```
vsip_cscalar_f vsip_cadd_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2);
```

```
void vsip_RCADD_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f* a3);
```

```
vsip_cscalar_f vsip_rcadd_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2);
```

Returns	Sum of input scalars if not void.
Argument a1	Input scalar.
Argument a2	Input scalar.
Argument a3	Output scalar (pointer) for void functions.

Scalar vector add

```
void vsip_svadd_f(
   vsip_scalar_f a1,
   vsip_vview_f* a2,
   vsip_vview_f* a3);
```

```
void vsip_csvadd_f(
   vsip_cscalar_f a1,
   vsip_cvview_f* a2,
   vsip_cvview_f* a3);
```

```
void vsip_rscvadd_f(
   vsip_scalar_f a1,
   vsip_cvview_f* a2,
   vsip_cvview_f* a3);
```

```
void vsip_svadd_i(
   vsip_scalar_i a1,
   vsip_vview_i* a2,
   vsip_vview_i* a3);
```

Argument a1 Input scalar.

Argument a2 Input vector.

Argument a3 Sum of scalar and vector elementwise.

Add two vectors element by element.

```
void vsip_vadd_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
void vsip_cvadd_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
void vsip_rcvadd_f(
   const vsip_vview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
void vsip_vadd_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
```

Argument a1 Input vector

Argument a2 Input vector

Argument a3 Sum of input vectors

alldestroy

Function to destroy a view and its associated block. If the block is bound to a user data array then the user data array is not destroyed.

```
void vsip_valldestroy_f(
   vsip_vview_f* a1);
void vsip_cvalldestroy_f(
   vsip_cvview_f* a1);
void vsip_valldestroy_i(
   vsip_vview_i* a1);
void vsip_valldestroy_vi(
   vsip_vview_vi* a1);
void vsip_valldestroy_mi(
   vsip_vview_mi* a1);
void vsip_valldestroy_bl(
   vsip_vview_bl* a1);
void vsip_malldestroy_f(
   vsip_mview_f* a1);
```

```
void vsip_cmalldestroy_f(
   vsip_cmview_f* a1);
```

Argument a1	The view to be destroyed.

alltrue

A boolean function returning true if all the elements in a boolean input view are true.

```
vsip_scalar_bl vsip_valltrue_bl(
   const vsip_vview_bl* a1);
```

Returns	False (0) if any of the elements in the input view are not false. Returns true (non-zero) if all the elements of the input view are true.
Argument a1	Input view.

and

Performs a bitwise “AND” operation between two integer views, or a logical “AND” between two boolean views.

```
void vsip_vand_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
void vsip_vand_bl(
   const vsip_vview_bl* a1,
   const vsip_vview_bl* a2,
   const vsip_vview_bl* a3);
```

Argument a1	Input view.
Argument a2	Input view.
Argument a3	Output view.

anytrue

A boolean function returning true if any of the elements in a boolean input view are true.

```
vsip_scalar_bl vsip_vanytrue_bl(
   const vsip_vview_bl* a1);
```

Returns	False (0) if all the elements in the input view are false. Returns true (non-zero) if any of the elements of the input view are true.
Argument a1	Input view.

arg

Scalar function to return the argument value (in radians) of a complex scalar.

```
vsip_scalar_f vsip_arg_f(
   vsip_cscalar_f a1);
```

Returns	The argument of the complex scalar.
Argument a1	Input complex scalar

asin

Inverse Sine function.

```
void vsip_asin_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

atan

Elementwise arctangent of a vector. This performs elementwise the atan function. For TASP this is just the ANSI C math functions, cast to the proper precision.

```
void vsip_vatan_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector of tangent values.

Argument a2 Output vector of arctangent values.

atan2

Elementwise arctangent of two vectors. For TASP VSIPL this is the same as the ANSI C math function atan2 cast to the proper precision.

```
void vsip_vatan2_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector denominator

Argument a2 Input vector numerator

Argument a3 Output vector of arctangent values of the quotient in radians.

bind

Used to create a (vector or matrix) view object and bind it to a block.

```
vsip_vview_f* vsip_vbind_f(
   const vsip_block_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_cvview_f* vsip_cvbind_f(
   const vsip_cblock_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_vview_i* vsip_vbind_i(
   const vsip_block_i *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_vview_vi* vsip_vbind_vi(
   const vsip_block_vi *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_vview_mi* vsip_vbind_mi(
   const vsip_block_mi *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_vview_bl* vsip_vbind_bl(
   const vsip_block_bl *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_mview_f* vsip_mbind_f(
   const vsip_block_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4,
   vsip_stride a5,
   vsip_length a6);
vsip_cmview_f* vsip_cmbind_f(
   const vsip_cblock_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4,
   vsip_stride a5,
   vsip_length a6);
```

Returns	A pointer to the view object created. Returns null on creation failure.
Argument a1	The block bound.
Argument a2	The offset from the beginning of the block where the view starts. Offsets are zero based and positive so that an offset of zero is the first element of the block.
Argument a3	A stride through the block. This indicates the number of elements in the block between vector view elements, or between matrix view elements in a column. A stride of zero will access only the element indicated by the offset, and a stride of one will access consecutive elements. A stride of N will access every Nth element. Strides may be negative indicating a direction of movement through the block opposite to that of a positive stride.
Argument a4	The length of the vector view in terms of elements or the length of a column (number of rows) in a matrix. The length is not zero based and a length of 1 indicates 1 element, and a length of N indicates N elements. The length is always greater than zero.
Argument a5	A stride through the block representing the row stride of a matrix. The row stride is the distance between consecutive elements in a row.
Argument a6	The row length (number of columns) of the matrix.

blackman

Create a unit stride zero offset floating point vector and fill it with a Blackman window of chosen length.

```
vsip_vview_f* vsip_vcreate_blackman_f(
   visp_length a1,
   vsip_memory_hint hint);
```

Returns	The created vector view filled with the window coefficients.
Argument a1	The number of window coefficients.
Argument a2	Memory hint. Not supported in TASP VSIPL. recommend placing a zero here.

blockadmit

Admit a block connected to user data for use by VSIPL functions. This function is used to change the state of a VSIPL user block from released to admitted.

```
int vsip_blockadmit_f(
   vsip_block_f* a1,
   vsip_scalar_bl a2);
int vsip_cblockadmit_f(
   vsip_cblock_f* a1,
   vsip_scalar_bl a2);
int vsip_blockadmit_i(
   vsip_block_i* a1,
   vsip_scalar_bl a2);
int vsip_blockadmit_vi(
   vsip_block_vi* a1,
   vsip_scalar_bl a2);
int vsip_blockadmit_mi(
   vsip_block_mi* a1,
   vsip_scalar_bl a2);
int vsip_blockadmit_bl(
   vsip_block_bl* a1,
   vsip_scalar_bl a2);
```

Returns	If the block admission succeeds 0 (zero) is returned. A nonzero value indicates a failure.
Argument a1	A block pointer for an instantiated (valid) block. The admission will fail if the block is bound to a null data pointer.
Argument a2	A boolean flag. True indicates the value of the data must be maintained during the state change.

blockbind

This function creates a block and binds it to a pointer to user allocated memory. The pointer defines the beginning of some user defined data array. It is the responsibility of the user to ensure the memory pointer has enough data allocated with it for the desired number of block elements.

```
vsip_block_f* vsip_blockbind_f(
   const vsip_scalar_f* a1,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_cblock_f* vsip_cblockbind_f(
   const vsip_scalar_f* a1,
   const vsip_scalar_f* a2,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_block_f* vsip_blockbind_i(
   const vsip_scalar_i* a1,
   vsip_length a3,
   vsip_memory_hint a4,);
vsip_block_vi* vsip_blockbind_vi(
   const vsip_scalar_vi* a1,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_block_mi* vsip_blockbind_mi(
   const vsip_scalar_vi* a1,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_block_bl* vsip_blockbind_bl(
   const vsip_scalar_bl* a1,
   vsip_length a3,
   vsip_memory_hint a4);
```

Returns	Pointer to created block.
Argument a1	Pointer to user defined data array. For complex blocks this pointer will point either to a single interleaved data array, or to the data array defined for real split data. For blocks of type matrix index the user data array is of type vector index. The matrix index is stored in an interleaved fashion, so the user data array has twice the number of vector index elements as the matrix index block created. It is permitted to bind to a NULL data pointer, but the block admission will fail until the block is rebound to a data pointer which is not NULL.
Argument a2	Pointer to user defined data array for imaginary complex data, if the split format is used, or to the null data pointer if interleaved complex is used.
Argument a3	Number of elements of the block type associated with the user data array(s).
Argument a4	This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

blockcreate

This function creates a block object. The block creation includes allocating memory for the data associated with the block

```
vsip_block_f* vsip_blockcreate_f(
   vsip_length a1,
   vsip_memory_hint a2);
```

```
vsip_cblock_f* vsip_cblockcreate_f
   vsip_length a1,
   vsip_memory_hint a2);
```

```
vsip_block_f* vsip_blockcreate_i(
   vsip_length a1,
   vsip_memory_hint a2);
```

```
vsip_block_vi* vsip_blockcreate_vi(
   vsip_length a1,
   vsip_memory_hint a2);
```

```
vsip_block_mi* vsip_blockcreate_mi(
   vsip_length a1,
   vsip_memory_hint a2);
```

```
vsip_block_bl* vsip_blockcreate_bl(
   vsip_length a1,
   vsip_memory_hint a2);
```

Returns	Pointer to created block.
Argument a1	Number of elements of the block type to be created and attached to the block. This is the block size, or the length of the block.
Argument a2	This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

blockdestroy

Destroy a block and any data bound to the block which was allocated by VSIPL. User data bound to the block is not destroyed.

```
void vsip_blockdestroy_f(
   vsip_block_f* a1);
```

```
void vsip_cblockdestroy_f(
   vsip_cblock_f* a1);
```

```
void vsip_blockdestroy_i(
   vsip_block_i* a1);
```

```
void vsip_blockdestroy_vi(
   vsip_block_vi* a1);
```

```
void vsip_blockdestroy_mi(
   vsip_block_mi* a1);
```

```
void vsip_blockdestroy_bl(
   vsip_block_bl* a1);
```

Argument a1	Block to be destroyed;

blockfind

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

```
vsip_scalar_f* vsip_blockfind_f(
   const vsip_block_f* a1);
```

```
void vsip_cblockfind_f(
   const vsip_cblock_f* a1,
   vsip_scalar_f* *a2,
   vsip_scalar_f* *a3);
vsip_scalar_i* vsip_blockfind_i(
   const vsip_block_i* a1);
vsip_scalar_vi* vsip_blockfind_vi(
   const vsip_block_vi* a1);
vsip_scalar_mi* vsip_blockfind_mi(
   const vsip_block_mi* a1);
vsip_scalar_bl* vsip_blockfind_bl(
   const vsip_block_bl* a1);
```

Returns	Pointer to the user data array bound to the block, or void for complex blocks. For VSIPL blocks (blocks not bound to user data) NULL is returned.
Argument a1	User released block
Argument a2	For complex, the data pointer to the user real data if split, or to the complex data if interleaved. For VSIPL complex blocks (blocks not bound to user data) NULL is returned
Argument a3	For complex, NULL if the user complex data is interleaved, and a pointer to the user imaginary data if split. For VSIPL complex blocks NULL is returned.

blockrebind

Bind an existing VSIPL user block to a new data array.

```
vsip_scalar_f* vsip_blockrebind_f(
   vsip_block_f* a1,
   const vsip_scalar_f* a2);
void vsip_cblockrebind_f(
   vsip_cblock_f* a1,
   const vsip_scalar_f* a2,
   const vsip_scalar_f* a3,
   vsip_scalar_f* *a4,
   vsip_scalar_f* *a5);
vsip_scalar_i* vsip_blockrebind_i(
   vsip_block_f* a1,
   const vsip_scalar_f* a2);
vsip_scalar_vi* vsip_blockrebind_vi(
   vsip_block_vi* a1,
   const vsip_scalar_vi* a2);
vsip_scalar_vi* vsip_blockrebind_mi(
   vsip_block_mi* a1,
   const vsip_scalar_vi* a2)
vsip_scalar_bl* vsip_blockrebind_bl(
   vsip_block_bl* a1,
   const vsip_scalar_bl* a2);
```

Returns	Except for complex, returns a pointer to the user data array bound to the block before the rebind. Returns void if the block is complex. If the block is not bound to a user data array, NULL is returned.

Argument a1	Pointer to block to be rebound.
Argument a2	Pointer to new data array to be bound to the user block. If the block is complex this data array is the real part of the complex number if the layout to be bound is split. Note that for blocks of type matrix index the data array is always interleaved.
Argument a3	A null pointer if the user complex data layout is interleaved, or a pointer to a data array encompassing the imaginary portion of the complex number if the data layout is split
Argument a4	A pointer to the previous real complex data array if the previous user complex data was split, or a pointer to the previous user interleaved complex data array. If the complex block is not bound to a user data array, NULL is returned.
Argument a5	A null pointer if the previous user data array was interleaved, or a pointer to the imaginary portion of the previous split complex user data array. If the complex block is not bound to a user data array, NULL is returned.

blockrelease

Release a user block. This function is used to change the state of a VSIPL user block from admitted to released.

```
vsip_scalar_f* vsip_blockrelease_f(
   vsip_block_f* a1,
   vsip_scalar_bl a2);

void vsip_cblockrelease_f(
   vsip_cblock_f * a1,
   vsip_scalar_bl a2,
   vsip_scalar_f* *a3
   vsip_scalar_f* *a4);

vsip_scalar_i* vsip_blockrelease_i(
   vsip_block_i* a1,
   vsip_scalar_bl a2);

vsip_scalar_vi* vsip_blockrelease_vi(
   vsip_block_vi* a1,
   vsip_scalar_bl a2);

vsip_scalar_vi* vsip_blockrelease_mi(
   vsip_block_mi* a1,
   vsip_scalar_bl a2);

vsip_scalar_bl* vsip_blockrelease_bl(
   vsip_block_bl* a1,
   vsip_scalar_bl a2);
```

Returns	Pointer to public data array, or void for complex. If the block is not bound to a user data array then NULL is returned.
Argument a1	Pointer to block to be released.
Argument a2	A boolean flag. True indicates the value of the data must be maintained during the state change.
Argument a3	For complex user data a pointer to the interleaved user data, or to the real part of the complex user data for split representation. If the block is not bound to a user data array then NULL is returned.

Argument a4	For complex a null data pointer for the interleaved representation, and a pointer to the imaginary data array for split representation. If the block is not bound to a user data array then NULL is returned.

cheby

Create a unit stride zero offset floating point vector and fill it with a Dolph-Chebyshev window of chosen length.

```
vsip_vview_f* vsip_vcreate_cheby_f(
   visp_length a1,
   vsip_scalar_f a2,
   vsip_memory_hint a3);
```

Returns	The created vector view filled with the window coefficients.
Argument a1	The number of window coefficients.
Argument a2	The desired window ripple in decibels below the main lobe.
Argument a3	Memory hint. Not supported in TASP VSIPL. recommend placing a zero here.

chold

Cholesky decomposition and solver for symmetric positive definite (SPD) linear system of the form $\bar{A}\vec{x}_i = \vec{b}_i$ where $\bar{A}$ is SPD and the set of vectors $\bar{X} = [\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_m]$ and $\bar{B} = [\vec{b}_0, \vec{b}_1, \ldots, \vec{b}_m]$ are solved.

Create a CHOLD object.

```
vsip_chol_f* vsip_chold_create_f(
   vsip_mat_uplo a1,
   vsip_length a2);
```

```
vsip_cchol_f* vsip_cchold_create_f(
   vsip_mat_uplo a1,
   vsip_length a2);
```

Returns	A CHOLD object for use by the Cholesky decomposition function set.
Argument a1	Since the matrix is symmetric only the upper or lower half need be referenced to compute the decomposition. This flag defines which half must be used.
Argument a2	Specifies the size of the input $N \mathrm{x} N$ matrix for which the object is created.

Compute a Cholesky decomposition and initialize the CHOLD object. The input matrix A is overwritten by the decomposition and associated with the CHOLD object. The matrix must not be modified or destroyed until after the CHOLD object is destroyed, or initialized with a different matrix.

```
int vsip_chold_f(
   vsip_chol_f* a1,
   const vsip_mview_f* a2);
```

```
int vsip_cchold_f(
   vsip_cchol_f* a1,
   const vsip_cmview_f* a2);
```

Returns A zero (0) if successful.

Argument a1 The CHOLD object to be initialized for matrix $\overline{A}$.

Argument a2 The input matrix $\overline{A}$. The argument a2 is overwritten and bound to the CHOLD object. It must not be modified until the decomposition is no longer needed.

Solve the SPD problem.

```
int vsip_cholsol_f(
   const vsip_chol_f* a1,
   const vsip_mview_f* a2);
```

```
int vsip_ccholsol_f(
   const vsip_cchol_f* a1,
   const vsip_cmview_f* a2);
```

Returns Zero (0) on success.

Argument a1 Initialized (for matrix $\overline{A}$) CHOLD object.

Argument a2 Input view of matrix $\overline{B}$, and output view of solution matrix $\overline{X}$.

Destroy the CHOLD object.

```
int vsip_chold_destroy_f(
   vsip_chold_f* a1);
```

```
int vsip_cchold_destroy_f(
   vsip_cchold_f* a1);
```

Returns Zero (0) on success.

Argument a1 CHOLD object to be destroyed.

Get public attributes of a CHOLD object.

```
void vsip_chold_getattr_f(
   const vsip_chol_f* chold,
   vsip_chol_f* attr);
```

```
void vsip_cchold_getattr_f(
   const vsip_cchol_f* chold,
   vsip_cchol_attr_f* attr);
```

Argument a1 Input CHOLD object.

Argument a2 Output CHOLD public attribute object.

cloneview

Creates a new view object with all the attributes of the parent object.

```
vsip_vview_f* vsip_vcloneview_f(
   const vsip_vview_f* a1);
```

```
vsip_cvview_f* vsip_cvcloneview_f(
   const vsip_cvview_f* a1);
```

```
vsip_vview_i* vsip_vcloneview_i(
   const vsip_vview_i* a1);
```

```
vsip_vview_vi* vsip_vcloneview_vi(
   const vsip_vview_vi* a1);

vsip_vview_mi* vsip_vcloneview_mi(
   const vsip_vview_mi* a1);

vsip_vview_bl* vsip_vcloneview_bl(
   const vsip_vview_bl* a1);

vsip_mview_f* vsip_mcloneview_f(
   const vsip_mview_f* a1);

vsip_cmview_f* vsip_cmcloneview_f(
   const vsip_cmview_f* a1);
```

Returns	A pointer to the new view.
Argument a1	The view to be cloned.

cmplx

Create a complex number or view from two real numbers or views.

Scalar complex.

```
vsip_cscalar_f vsip_cmplx_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2);

void vsip_CMPLX_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2,
   vsip_scalar_f* a3);
```

Returns	For non-void scalar the complex output scalar.
Argument a1	An input scalar representing the real part.
Argument a2	An input scalar representing the imaginary part.
Argument a3	For void scalar the complex output scalar.

Vector complex.

```
void vsip_vcmplx_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2,
   vsip_cvview_f* a3);
```

Argument a1	Input vector representing the real part
Argument a2	Input vector representing the imaginary part.
Argument a3	The complex output vector.

clip

Given upper and lower comparison threshold values, compare against a view element-wise. If the view elements are not less than the upper threshold or greater than the lower threshold output the view element. For view elements not greater than the lower comparison threshold replace the output element with a lower threshold replacement value, and if the view elements are greater than the upper comparison threshold replace the view element with the upper replacement value. The order of comparison is the value less than or equal to the lower comparison threshold, then is the value less than the upper comparison value. If neither condition is met then the value is greater than or equal to the upper comparison threshold value. Once a condition is met, the rule is followed and no other comparisons are done. There is no requirement that the upper values be greater than the lower values. The terms upper and lower only imply the argument order and the comparison and replacement done.

$$\begin{gathered} a1 \le a2 \Rightarrow a6 = a4 \\ \text{else} \\ a1 < a3 \Rightarrow a6 = a1 \\ \text{else} \\ a6 = a5 \end{gathered}$$

```
void vsip_vclip_f(
   const vsip_vview_f *a1,
   vsip_scalar_f a2,
   vsip_scalar_f a3,
   vsip_scalar_f a4,
   vsip_scalar_f a5,
   const vsip_vview_f *a6);
void vsip_vclip_i(
   const vsip_vview_i *a1,
   vsip_scalar_i a2,
   vsip_scalar_i a3,
   vsip_scalar_i a4,
   vsip_scalar_i a5,
   const vsip_vview_i *a6);
```

Argument a1	Input vector.
Argument a2	Lower comparison threshold.
Argument a3	Upper comparison threshold.
Argument a4	Lower threshold replacement value.
Argument a5	Upper threshold replacement value.
Argument a6	Output vector.

cmagsq

Find the complex magnitude squared.

Scalar complex magnitude squared.

```
vsip_scalar_f vsip_cmagsq_f(
   vsip_cscalar_f a1);
```

Returns	Magnitude squared value of complex scalar.
Argument a1	Input complex scalar.

Vector complex magnitude squared. For a complex vector find the magnitude squared value of each element.

```
void vsip_vcmagsq_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

cmaxmgsq

Complex maximum magnitude squared comparison. Compare the magnitude squared values of two complex vectors elementwise and output the maximum magnitude squared of each element comparison into an output vector.

```
void vsip_vcmaxmgsq_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector.

Argument a2 Input vector.

Argument a3 Output vector.

cmaxmagsqval

Maximum magnitude squared value of a complex vector. Find the maximum magnitude squared value among all the elements of a complex view.

```
vsip_scalar_f vsip_vcmaxmagsqval(
   vsip_cvview_f* a1,
   vsip_index a2);
```

Returns The maximum magnitude squared value.

Argument a1 The input vector.

Argument a2 The index (into the input vector) of the selected value with the maximum magnitude squared.

cminmgsq

Complex minimum magnitude squared comparison. Compare the magnitude squared values of two complex vectors elementwise and output the minimum magnitude squared of each element comparison into an output vector.

```
void vsip_vcminmgsq_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector.

Argument a2 Input vector.

Argument a3 Output vector.

cminmagsqval

Minimum magnitude squared value of a complex vector. Find the minimum magnitude squared value among all the elements of a complex view.

```
vsip_scalar_f vsip_vcminmagsqval(
   vsip_cvview_f* a1,
   vsip_index a2);
```

Returns	The minimum magnitude squared value.
Argument a1	The input vector.
Argument a2	The index (into the input vector) of the selected value with the minimum magnitude squared.

copy

The copy function copies data from one view to another view. This function is also used to convert data types, for instance from integer to float.

```
void vsip_vcopy_f_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);

void vsip_vcopy_f_i(
   const vsip_vview_f* a1,
   const vsip_vview_i* a2);

void vsip_vcopy_i_f(
   const vsip_vview_i* a1,
   const vsip_vview_f* a2);

void vsip_cvcopy_f_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);

void vsip_vcopy_i_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2);

void vsip_vcopy_vi_i(
   const vsip_vview_vi* a1,
   const vsip_vview_i* a2);

void vsip_vcopy_vi_vi(
   const vsip_vview_vi* a1,
   const vsip_vview_vi* a2);

void vsip_vcopy_i_vi(
   const vsip_vview_i* a1,
   const vsip_vview_vi* a2);

void vsip_vcopy_mi_mi(
   const vsip_vview_mi* a1,
   const vsip_vview_mi* a2);

void vsip_vcopy_bl_bl(
   const vsip_vview_bl* a1,
   const vsip_vview_bl* a2);
```

```
void vsip_vcopy_bl_f(
   const vsip_vview_bl* a1,
   const vsip_vview_f* a2);
```

```
void vsip_vcopy_f_bl(
   const vsip_vview_f* a1,
   const vsip_vview_bl* a2);
```

```
void vsip_mcopy_f_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2);
```

```
void vsip_cmcopy_f_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2);
```

Argument a1	Input to be copied.
Argument a2	Output, a copy of the input with possibly a data type conversion. Note that when copying boolean to float false values are copied as zero, and true values are copied as one.

colview

Create a vector view of a selected column of a matrix

```
vsip_vview_f* vsip_mcolview_f(
   const vsip_mview_f* a1,
   vsip_index a2);
```

```
vsip_cvview_f* vsip_cmcolview_f(
   const vsip_cmview_f* a1,
   vsip_index a2);
```

Returns	A vector view of the selected column, or a NULL if the memory allocation for the view object fails.
Argument a1	Input view.
Argument a2	Index of desired view. Indices are zero based so that the first (left most) column of the matrix has index zero.

conj

Conjugate a complex scalar.

```
vsip_cscalar_f vsip_conj_f(
   vsip_cscalar_f a1);
```

```
void vsip_CONJ_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f* a2);
```

Conjugate a complex vector.

```
void vsip_cvconj_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1	Input.

Argument a2 Output.

convolve

Calculate a convolution of a filter kernel and a view.

Create a convolution object.

```
vsip_conv1d_f* vsip_conv1d_create_f(
   const vsip_vview_f* a1,
   vsip_symmetry a2,
   vsip_length a3,
   vsip_length a4,
   vsip_support_region a5,
   vsip_length a6,
   vsip_alg_hint a7);
```

Returns	A convolution object.
Argument a1	A view containing the kernel information. The view will either contain all the kernel information, in which case the symmetry argument below is nonsym, or only the non-redundant part of the kernel, in which case the symmetry argument will determine the length of the kernel.
Argument a2	The symmetry argument. NONSYM implies that all the kernel coefficients are in the kernel argument. SYM_EVEN_LEN_ODD implies the kernel is of odd length and symmetric in which case the first half $(N-1)/2+1$ kernel coefficients are included in the kernel argument. If the symmetry argument is SYM_EVEN_LEN_EVEN then the kernel is of even length and symmetric, and only the first $N/2$ coefficients are included in the kernel argument.
Argument a3	The length of the input vector expected when convolving with the kernel.
Argument a4	A decimation factor.
Argument a5	The region of support of the output. For FULL support with decimation D the length of the output will be the $\operatorname{floor}((N+M-2)/D)+1$ where M is the length of the kernel and N is the length of the input vector. For SAME support the length will be $\operatorname{floor}((N-1)/D)+1$, and for MIN support the length will be $(\operatorname{floor}((N-1)/D)-\operatorname{floor}((M-1)/D))+1$.
Argument a6	Number of times the function will be called. This is not supported in TASP VSIPL. Recommend placing a zero here, although any number will do.
Argument a7	VSIPL algorit.him hint. Not supported in TASP VSIPL. Recommend placing a zero here, although any valid hint will do.

Calculate the convolution.

```
void vsip_convolve1d_f(
   const vsip_conv1d_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Convolution argument.

Argument a2 Input view. The size of the input view must agree with the size given in the convolution object.

Argument a3 Output view. The size of the output view must agree with the required size given the decimation factor, and the size of the kernel and the input vector. See argument a5 for the convolution creation function.

Destroy the convolution object.

```
int vsip_conv1d_destroy_f(
   vsip_conv1d_f* conv1d);
```

Returns A zero on success, non-zero on failure.

Argument a1 Convolution object to be destroyed.

Get the attributes of the convolution object.

```
void vsip_conv1d_getattr_f(
   const vsip_conv1d_f* a1,
   vsip_conv1d_attr_f* a2);
```

Argument a1 Input convolution object.

Argument a2 The pointer to the attributes. The kernel_len is the total number of coefficients in the input filter, the data_len is the length of the input vector, the out_len is the length of the output vector.

correlate

Correlate two views.

Create correlation object.

```
vsip_corr1d_f* vsip_corr1d_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_support_region a3,
   vsip_length a4,
   vsip_alg_hint a5);
```

```
vsip_ccorr1d_f* vsip_ccorr1d_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_support_region a3,
   vsip_length a4,
   vsip_alg_hint a5);
```

Returns Correlation object.

Argument a1 Length of input reference view.

Argument a2 Length of input data view. Must be greater than or equal to the reference view length.

Argument a3 Region of support. This works the same as the convolution except that there is no decimation, so D is one. The length of the reference view replaces the length of the kernel. So for FULL the length is $N + M - 1$, for SAME the length is N and for MIN the length is $N - M + 1$.

Argument a4	Number of times the object is expected to be used. Not supported in TASP VSIPL. Recommend placing a zero (0) here, although any number will work.
Argument a5	Algorithm hint. Not supported in TASP VSIPL. Recommend placing a zero here, although any valid hint will work.

Correlate two views.

```
void vsip_correlate1d_f(
   const vsip_corr1d_f* a1,
   vsip_bias a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4,
   const vsip_vview_f* a5);
```

```
void vsip_ccorrelate1d_f(
   const vsip_ccorr1d_f* a1,
   vsip_bias a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4,
   const vsip_cvview_f* a5);
```

Argument a1	Correlation object.
Argument a2	Type of correlation, biased or unbiased. Biased implies the correlation is done with no normalization factor. This means that the tails will be biased with respect to the middle portion where the entire reference view is overlapped with the data view. Unbiased implies that the length of the overlap of the calculated correlation value for a particular lag will be used to normalize the value.
Argument a3	Input reference view.
Argument a4	Input data view.
Argument a5	Output view of correlation values.

Destroy correlation object

```
int vsip_corr1d_destroy_f(
   vsip_corr1d_f* a1);
```

```
int vsip_ccorr1d_destroy_f(
   vsip_ccorr1d_f* a1);
```

Returns	Zero (0) on success, non-zero on failure.
Argument a1	Correlation object to be destroyed.

Get public attributes from a correlation object.

```
void vsip_corr1d_getattr_f(
   vsip_corr1d_f* a1,
   vsip_corr1d_attr_f* a2);
```

```
void vsip_ccorr1d_getattr_f(
   vsip_ccorr1d_f* a1,
   vvsip_ccorr1d_attr_f* a2);
```

Argument a1	Input correlation object.
Argument a2	Output attribute object. The ref_len, and the data_len are the reference view and data view lengths respectively. The lag_len is the length of the output view.

cos

Elementwise Cosine of a vector.

```
void vsip_vcos_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1	Input vector of angles in radian format.
Argument a2	Output vector of Cosine values.

covsol

Covariance Solver. Solve a set of equations

$$\bar{A}^T\bar{A}\vec{x}_i = \vec{b}_i$$

or

$$\bar{A}^H\bar{A}\vec{x}_i' = \vec{b}_i$$

for the input vector set $\bar{B}$ and output vector set $\bar{X}$

$$\vec{X} = [\vec{x}_o, \vec{x}_1, \ldots, \vec{x}_m]$$

$$\vec{B} = [\vec{b}_o, \vec{b}_1, \ldots, \vec{b}_m]$$

```
int vsip_covsol_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2);
int vsip_ccovsol_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2);
```

Returns	Zero (0) on success, minus one (-1) on failure due to a memory allocation problem. Positive return indicates failure for some other reason.
Argument a1	Input matrix $\bar{A}$
Argument a2	Input matrix $\bar{B}$, output matrix $\bar{X}$.

create

Convenience function to create a view, the block and the data associated with the block all at the same time. The created view accesses the entire block. For a vector view this means an offset of zero, a stride of one, and a length equal to the block size. For a matrix view the block size is the product of the row length and the column length. The stride in the selected major direction will be one, and the stride in the other direction will be equal to the length of the major direction axis. For instance a row major matrix will have a row stride of one, and a column stride equal to the row length.

```
vsip_vview_f* vsip_vcreate_f(
   vsip_length a1,
   vsip_memory_hint a3);

vsip_cvview_f* vsip_cvcreate_f(
   vsip_length a1,
   vsip_memory_hint a3);

vsip_vview_i* vsip_vcreate_i(
   vsip_length a1,
   vsip_memory_hint a4);

vsip_vview_vi* vsip_vcreate_vi(
   vsip_length a1,
   vsip_memory_hint a4);

vsip_vview_mi* vsip_vcreate_mi(
   vsip_length a1,
   vsip_memory_hint a4);

vsip_vview_bl* vsip_vcreate_bl(
   vsip_length a1,
   vsip_memory_hint a4);

vsip_mview_f* vsip_mcreate_f(
   vsip_length a1,
   vsip_length a2,
   vsip_major a3,
   vsip_memory_hint a4);

vsip_cmview_f* vsip_cmcreate_f(
   vsip_length a1,
   vsip_length a2,
   vsip_major a3,
   vsip_memory_hint a4);
```

Returns	Pointer to vector view requested.
Argument a1	Length of the vector or matrix major direction.
Argument a2	Length of the matrix minor direction.
Argument a3	Enumerated type indicating major direction.
Argument a4	This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

cstorage

Indicates the preferred method of complex storage for user data in a particular VSIPL implementation.

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

Returns	A value based on the enumerated typedef

destroy

Function to destroy a view.

```
vsip_block_f* vsip_vdestroy_f(
   vsip_vview_f* a1);
```

```
vsip_cblock_f* vsip_cvdestroy_f(
   vsip_cvview_f* a1);
vsip_block_i* vsip_vdestroy_i(
   vsip_vview_i* a1);
vsip_block_vi* vsip_vdestroy_vi(
   vsip_vview_vi* a1);
vsip_block_mi* vsip_vdestroy_mi(
   vsip_vview_mi* a1);
vsip_block_bl* vsip_vdestroy_bl(
   vsip_vview_bl* a1);
vsip_block_f* vsip_mdestroy_f(
   vsip_mview_f* a1);
vsip_cblock_f* vsip_cmdestroy_f(
   vsip_cmview_f* a1);
```

`Returns`	A pointer to the block the view was bound to.
`Argument a1`	The pointer to the view to be destroyed.

diagview

Create a view of a selected diagonal of a matrix.

```
vsip_vview_f* vsip_mdiagview_f(
   const vsip_mview_f* a1,
   vsip_stride a2);
vsip_cvview_f* vsip_cmdiagview_f(
   const vsip_cmview_f* a1,
   vsip_stride a2);
```

`Returns`	A pointer to the view of the selected diagonal.
`Argument a1`	Input matrix view.
`Argument a2`	The index of the selected view. An index of zero is the main diagonal with the first element of the created view being the first element in the input matrix. A negative value selects, in order, the diagonals below the main diagonal, and a positive value selects, in order, the diagonals above the main diagonal.The index argument has a type of stride because the standard VSIPL index is some type of unsigned int. VSIPL indices are zero based, so this is not an index in the standard VSIPL sense, and is defined with a stride type to meet the requirements of a negative index.

div

Divide two scalars.

```
vsip_cscalar_f vsip_cdiv_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2);
vsip_cscalar_f vsip_crdiv_f(
   vsip_csclar_f a1,
   vsip_scalar_f a2);
void vsip_CDIV_f(
   vsip_cscalar_f a1,
      vsip_cscalar_f a2,
      vsip_cscalar_f *a3);
```

```
   void vsip_CDIV_f(
      vsip_cscalar_f a1,
      vsip_scalar_f a2,
      vsip_cscalar_f *a3);
```

Divide two vectors element by element.

```
   void vsip_vdiv_f(
      const vsip_vview_f* a1,
      const vsip_vview_f* a2,
      const vsip_vview_f* a3);
```

```
   void vsip_cvdiv_f(
      const vsip_cvview_f* a1,
      const vsip_cvview_f* a2,
      const vsip_cvview_f* a3);
```

Scalar vector divide.

```
   void vsip_svdiv_f(
      vsip_scalar_f a1,
      vsip_vview_f* a2,
      vsip_vview_f* a3);
```

Returns	For non void scalar function the quotient.
Argument a1	The numerator input.
Argument a2	The denominator output.
Argument a3	The quotient output.

dot

Dot products. A dot product is an elementwise multiply of two vectors with a sum of the resulting vector.

Real Dot Product

```
   vsip_scalar_f vsip_vdot_f(
      const vsip_vview_f* a1,
      const vsip_vview_f* a2);
```

Returns	Dot Product value.
Argument a1	Real input vector.
Argument a2	Real input vector.

Complex Dot Product

```
   vsip_cscalar_f vsip_cvdot_f(
      const vsip_cvview_f* a1,
      const vsip_cvview_f* a2);
```

Returns	Dot Product value.
Argument a1	Complex input vector.
Argument a2	Complex input vector.

Complex Conjugate Dot Product. The dot product here is done between the first input vector and the complex conjugate of the second input vector.

```
vsip_cscalar_f vsip_cvjdot_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Returns	Dot Product value.
Argument a1	Complex input vector.
Argument a2	Complex input vector.

euler

Euler function. Elementwise compute the Sine value and the Cosine value of an input vector. Place the Cosine value in the real part of a complex output vector, and place the Sine value in the imaginary part of a complex output vector

```
void vsip_veuler_f(
   vsip_vview_f *a1,
   vsip_cvview_f *a2);
```

Argument a1	Input vector.
Argument a2	Output vector.

exp

Natural (base e) exponential functions

Scalar natural (base e) exponential.

```
vsip_cscalar_f vsip_cexp_f(
   vsip_cscalar_f a1);
```

```
void vsip_CEXP_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f* a2);
```

Elementwise natural (base e) exponential of a vector.

```
void vsip_vexp_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvexp_v(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Returns	For non void scalar the exponential of the argument.
Argument a1	Input
Argument a2	Output

exp10

Base 10 exponential functions.

```
void vsip_vexp10_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector of base 10 exponentials.

expoavg

Exponential average function. Compute a weighted average elementwise of two vectors. $\vec{a3} = a1\vec{a2} + (1 - a1)\vec{a3}$

```
void vsip_vexpoavg_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

```
void vsip_cvexpoavg_f(
   vsip_scalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Real input scalar weighting factor.

Argument a2 Input view.

Argument a3 Input and output view.

fft

Compute a Discrete Fourier Transform using FFT methods for radices of 2 and at least one factor of 3 as a minimum. The current TASP VSIPL FFT uses building block factors of 2, 4, 8, 3, 5,and 7.

Create an FFT object. There is only one type of FFT object for one dimensional FFTs. The FFT object maintains state information to determine which type FFT it is created for. It is the responsibility of the user to keep track of type FFT the object was created for.

Create an FFT object for doing a complex to complex out of place FFT.

```
vsip_fft_f* vsip_ccfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   vsip_fft_dir a3,
   unsigned int a4,
   vsip_alg_hint a5);
```

Create an FFT object for doing a complex to complex in place FFT.

```
vsip_fft_f* vsip_ccfftip_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   vsip_fft_dir a3,
unsigned int a4,
vsip_alg_hint a5);
```

Create an FFT object for doing a real to complex out of place FFT. All real to complex FFT objects are created to go in the forward direction.

```
vsip_fft_f* vsip_rcfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   unsigned int a4,
   vsip_alg_hint a5);
```

Create an FFT object for doing a complex to real out of place FFT. All complex to real FFT objects are created to go in the inverse direction.

```
vsip_fft_f* vsip_crfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   unsigned int a4,
   vsip_alg_hint a5);
```

Returns	FFT object useful for creating a (user selected direction) forward or inverse FFT, or null on creation failure.
Argument a1	Length of FFT. Except for the complex to real FFT object this will be the length of the input vector to the FFT. For the complex to real FFT object this is the length of the output vector. For the real to complex and complex to real FFTs the FFT length must be even.
Argument a2	A scale factor. If a scale factor of 1 is used for a forward FFT then a scale factor of $1/(a1)$ in the inverse FFT will get back the original vector.
Argument a3	An enumerated type defining the direction of the FFT. You may use VSIP_FFT_FWD (-1) for the forward FFT and VSIP_FFT_INV (+1) for the inverse FFT. The direction of the complex to real and real to complex FFT is hard coded in the algorithm, and this argument is not included.
Argument a4	Estimated number of times the object will be used in an FFT call. This option is not supported in TASP VSIPL. Recommend placing zero here, although any number will work.
Argument a5	This option is not supported in TASP VSIPL. Recommend placing a zero here, although any valid algorithm hint will work.

Do the FFT. The FFT object must be created with the proper creation function to match the FFT function.

Complex to complex out of place FFT.

```
void vsip_ccfftop_f(
   const vsip_fft_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Complex to complex in place FFT.

```
void vsip_ccfftip_f(
   const vsip_fft_f* a1,
   const vsip_cvview_f* a2);
```

Real to complex out of place FFT. The input vector of length N must be even. The complex output vector is of length $N/2+1$.

```
void vsip_rcfftop_f(
   const vsip_fft_f* a1,
   const vsip_vview_f* a2,
   const vsip_cvview_f* a3);
```

Complex to real out of place FFT. The output vector of length N must be even. The complex input vector is of length $N/2+1$.

```
void vsip_crfftop_f(
   const vsip_fft_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1	FFT object.
Argument a2	Input data, and output data for in place FFT.
Argument a3	Output data

Destroy an FFT object.

```
int vsip_fft_destroy_f(
   vsip_fft_f* a1);
```

Returns	zero on success.
Argument a1	FFT object to be destroyed.

Get the attributes of an FFT object. The attribute object contains the *input* data size, the *output* data size, whether the object was created for in place or out of place use, the scale factor, and the direction of the FFT object.

```
void vsip_fft_getattr_f(
   const vsip_fft_f* a1,
   vsip_fft_attr_f* a2);
```

Argument a1	FFT object.
Argument a2	FFT attribute object.

fftm

Compute a Discrete Fourier Transform Multiple times using FFT methods for radices of 2 and at least one factor of 3 as a minimum. The current TASP VSIPL Multiple FFT uses building block factors of 2, 4, 8, 3, 5,and 7.

Create a Multiple FFT object. There is only one type of Multiple FFT object for one dimensional FFTs. The Multiple FFT object maintains state information to determine which type Multiple FFT it is created for. It is the responsibility of the user to keep track of the Multiple FFT function the object was created for.

Create a Multiple FFT object for doing a complex to complex out of place FFT.

```
vsip_fftm_f* vsip_ccfftmop_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_scalar_f a3,
   vsip_fft_dir a4,
   vsip_major a5
   unsigned int a6,
   vsip_alg_hint a7);
```

Create a Multiple FFT object for doing a complex to complex in place FFT.

```
vsip_fftm_f* vsip_ccfftmip_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_scalar_f a3,
   vsip_fft_dir a4,
   vsip_major a5,
   unsigned int a6
   vsip_alg_hint a7);
```

Create a Multiple FFT object for doing a real to complex out of place FFT. All real to complex Multiple FFT objects are created to go in the forward direction.

```
vsip_fftm_f* vsip_rcfftmop_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_scalar_f a3,
   vsip_major a5,
   unsigned int a6,
   vsip_alg_hint a7);
```

Create a Multiple FFT object for doing a complex to real out of place FFT. All complex to real Multiple FFT objects are created to go in the inverse direction.

```
vsip_fftm_f* vsip_crfftmop_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_scalar_f a3,
   vsip_major a5,
   unsigned int a6,
   vsip_alg_hint a7);
```

`Returns`	FFT Multiple object useful for creating a (user selected direction) forward or inverse FFT, or null on creation failure.
`Argument a1`	If the *selected direction* for doing the FFT is along the *column*, then this is the length of the FFT. For the complex to complex cases, and the real to complex case this is the number of rows in the input matrix (column length). For the complex to real FFT multiple object this is the length of the output matrix column. For the real to complex and complex to real FFTs this value must be even if the column direction is selected.

	If the *selected direction* for doing the FFT is along the *row*, then this is the number of FFTs to be done (the number of rows in the input matrix)
Argument a2	If the *selected direction* for doing the FFT is along the *row*, then this is the length of the FFT. For the complex to complex cases, and the real to complex case this is the number of columns in the input matrix (row length). For the complex to real FFT Multiple object this is the length of the output matrix row. For the real to complex and complex to real FFTs this value must be even if the row direction is selected. If the *selected direction* for doing the FFT is along the *column*, then this is the number of FFTs to be done (the number of columns in the input matrix)
Argument a3	A scale factor. If a scale factor of 1 is used for a forward FFT then a scale factor of $1/(a1)$ in the inverse FFT will get back the original vector.
Argument a4	An enumerated type defining the direction of the FFT. You may use VSIP_FFT_FWD (-1) for the forward FFT and VSIP_FFT_INV (+1) for the inverse FFT. The direction of the complex to real and real to complex FFT is hard coded in the algorithm, and this argument is not included.
Argument a5	The direction along which the FFT Multiple will be done. The length of the other direction is the number of FFTs done.
Argument a6	Estimated number of times the object will be used in an FFT Multiple call. This option is not supported in TASP VSIPL. Recommend placing zero here, although any number will work.
Argument a7	This option is not supported in TASP VSIPL. Recommend placing a zero here, although any valid algorithm hint will work.

Do the Multiple FFT. The FFT Multiple object must be created with the proper creation function to match the FFT Multiple function.

Complex to complex out of place Multiple FFT.

```
void vsip_ccfftmop_f(
   const vsip_fftm_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cmview_f* a3);
```

Complex to complex in place Multiple FFT.

```
void vsip_ccfftmip_f(
   const vsip_fftm_f* a1,
   const vsip_cmview_f* a2);
```

Real to complex out of place Multiple FFT. The input matrix must be even length (L) along the major direction. The complex output matrix is of length $L/2 + 1$ along the major direction.

```
void vsip_rcfftmop_f(
   const vsip_fftm_f* a1,
   const vsip_mview_f* a2,
   const vsip_cmview_f* a3);
```

Complex to real out of place Multiple FFT. The output matrix must be even length L along the major direction. The complex input vector is of length $L/2 + 1$.

```
void vsip_crfftmop_f(
   const vsip_fftm_f* a1,
   const vsip_cmview_f* a2,
   const vsip_mview_f* a3);
```

Argument a1	Multiple FFT object.
Argument a2	Input view for all Multiple FFTs, and output view for in place Multiple FFT.
Argument a3	Output view.

Destroy a Multiple FFT object.

```
int vsip_fftm_destroy_f(
   vsip_fftm_f* a1);
```

Returns	zero on success.
Argument a1	Multiple FFT object to be destroyed.

Get the attributes of a Multiple FFT object. The attribute object contains the *input* data size, the *output* data size, whether the object was created for in place or out of place use, the scale factor, the major direction, and the direction of the Multiple FFT object.

```
void vsip_fftm_getattr_f(
   const vsip_fftm_f* a1,
   vsip_fftm_attr_f* a2);
```

Argument a1	Multiple FFT object.
Argument a2	Multiple FFT attribute object.

fill

Fill a vector with a constant value.

```
void vsip_vfill_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvfill_f(
   vsip_cscalar_f a1,
   const vsip_cvview_f* a2);
```

```
void vsip_vfill_i(
   vsip_scalar_i a1,
   const vsip_vview_i* a2);
```

Argument a1	Scalar value to fill output vector with.
Argument a2	Output vector

fir

Finite impulse response filter with decimation.

Finite impulse response filter object create.

```
vsip_fir_f* vsip_fir_create_f(
   const vsip_vview_f* a1,
   vsip_symmetry a2,
   vsip_length a3,
   vsip_length a4,
   vsip_obj_state a5,
   unsigned int a6,
   vsip_alg_hint a7);
```

```
vsip_cfir_f* vsip_cfir_create_f(
   const vsip_cvview_f* a1,
   vsip_symmetry a2,
   vsip_length a3,
   vsip_length a4,
   unsigned int a5,
   vsip_alg_hint a6);
```

Returns	Pointer to FIR object.
Argument a1	Vector view containing filter kernel. If a1 holds all the filter coefficients then **VSIP_NONYSM** is proper for a2. If the filter coefficients are symmetric and the number of coefficients is even then only the first half of the coefficients are necessary in a1 and **VSIP_SYM_EVEN_LEN_EVEN** is proper for a2. If the filter coefficients are symmetric and the number of coefficients is odd then only the first half of the coefficients plus the center coefficient are necessary in a1 and **VSIP_SYM_EVEN_LEN_ODD** is proper for a2.
Argument a2	Symmetry enumerated typedef associated with the selected kernel
Argument a3	Length of the data to be filtered at a time.
Argument a4	Decimation factor.
Argument a5	Enumerated type indicating if the object state should be saved between function calls to **vsip_firflt_f**. To save state use **VSIP_STATE_SAVE**. To ignore state use **VSIP_STATE_NO_SAVE**.
Argument a6	Estimated number of times the object will be used. Not implemented in TASP VSIPL. Recommend placing a 0 (zero) in this spot.
Argument a7	Algorithm hint. Not implemented in TASP VSIPL. Recommend placing a 0 (zero) in this spot.

Finite impulse response filter function.

```
int vsip_firflt_f(
   vsip_fir_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

```
int vsip_cfirflt_f(
   vsip_cfir_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Returns	The number of output samples placed in argument a3.
Argument a1	A FIR filter object
Argument a2	The input vector to be filtered

`Argument a3`	The output vector. The length of the output vector must be the quotient of the length of the input vector divided by the decimation. The quotient is rounded up to give the ceiling of the division.

FIR filter object destruction function.

```
int vsip_fir_destroy_f(
   vsip_fir_f* a1);
```

```
int vsip_cfir_destroy_f(
   vsip_cfir_f* a1);
```

`Returns`	Returns 0 (zero) on success.
`Argument a1`	The FIR filter object to be destroyed.

FIR filter object get attributes function.

```
void vsip_fir_getattr_f(
   const vsip_fir_f* a1,
   vsip_fir_attr_f* a2);
```

```
void vsip_cfir_getattr_f(
   const vsip_cfir_f* a1,
   vsip_cfir_attr_f* a2);
```

`Argument a1`	The FIR filter object.
`Argument a2`	The FIR attribute object.

gather

A vector of indices is used elementwise to index an input view. The indexed values are placed, elementwise, in an output vector view. The vector of indices, and the output vector view have the same length, and are indexed the same. The only requirement on the input view is that the index vector contain valid entrees to index the input view. For the core profile only vector views are defined for input.

```
void vsip_vgather_f(
   const vsip_vview_f* a1,
   const vsip_vview_vi* a2,
   const vsip_vview_f* a3);
```

```
void vsip_cvgather_f(
   const vsip_cvview_f* a1,
   const vsip_vview_vi* a2,
   const vsip_cvview_f* a3);
```

```
void vsip_vgather_i(
   const vsip_vview_i* a1,
   const vsip_vview_vi* a2,
   const vsip_vview_i* a3);
```

`Argument a1`	Input view.
`Argument a2`	Input vector view of indices (index vector).
`Argument a3`	Output vector view.

gemp

General matrix product. The general matrix product operates on three matrices $\overline{A}$, $\overline{B}$, and $\overline{C}$ and two scalars α and β in conjunction with a enumerated type to indicate matrix unary matrix operations (normal, transpose, hermitian or conjugation) on input matrices $\overline{A}$ and $\overline{B}$ to produce a general matrix product of the following form.

$$\overline{C} = \alpha \mathrm{op}(\overline{A})\mathrm{op}(\overline{B}) + \beta\overline{C}$$

The size of the matrices must be such that the selected operations will work using normal linear algebra methods.

```
void vsip_gemp_f(
   vsip_scalar_f a1,
   const vsip_mview_f *a2,
   vsip_mat_op a3,
   const vsip_mview_f *a4,
   vsip_mat_op a5,
   vsip_scalar_f a6,
   vsip_mview_f* a7);
void vsip_cgemp_f(
   vsip_cscalar_f a1,
   const vsip_cmview_f *a2,
   vsip_mat_op a3,
   const vsip_cmview_f *a4,
   vsip_mat_op a5,
   vsip_cscalar_f a6,
   vsip_cmview_f* a7);
```

Argument a1 Scalar multiplier α.

Argument a2 Input matrix $\overline{A}$.

Argument a3 Unary matrix operation on a2 before matrix multiply.

Argument a4 Input matrix $\overline{B}$

Argument a5 Unary matrix operation on a3 before matrix multiply.

Argument a6 Scalar multiplier β.

Argument a7 Input/output matrix $\overline{C}$.

gems

General matrix sum. The general matrix sum operates on two matrices $\overline{A}$ and $\overline{B}$ using a unary matrix operator on matrix $\overline{A}$ and multiplying matrix $\overline{A}$ and $\overline{B}$ by scalars α and β before summing the results.

$$\overline{B} = \alpha \mathrm{op}(\overline{A}) + \beta\overline{B}$$

The matrices $\overline{A}$ and $\overline{B}$ must be sized properly so that the matrix sum may be done.

```
void vsip_gems_f(
   vsip_scalar_f a1,
   const vsip_mview_f *a2,
   vsip_mat_op a3,
   vsip_scalar_f a4,
   const vsip_mview_f *a5);
```

```
void vsip_cgems_f(
   vsip_cscalar_f a1,
   const vsip_cmview_f *a2,
   vsip_mat_op a3,
   vsip_cscalar_f a4,
   const vsip_cmview_f *a5);
```

Argument a1 Scalar multiplier α.

Argument a2 Input matrix view A.

Argument a3 Matrix operation to perform on a2.

Argument a4 Scalar multiplier β.

Argument a5 Input/Output matrix view B.

get

Get an element from a view

```
vsip_scalar_f vsip_vget_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2);
```

```
vsip_cscalar_f vsip_cvget_f(
   const vsip_cvview_f* a1,
   vsip_scalar_vi a2);
```

```
vsip_scalar_i vsip_vget_i(
   const vsip_vview_i* a1,
   vsip_scalar_vi a2);
```

```
vsip_scalar_vi vsip_vget_vi(
   const vsip_vview_vi* a1,
   vsip_scalar_vi a2);
```

```
vsip_scalar_mi vsip_vget_mi(
   const vsip_vview_mi* a1,
   vsip_scalar_vi a2);
```

```
vsip_scalar_bl vsip_vget_bl(
   const vsip_vview_bl* a1,
   vsip_scalar_vi a2);
```

```
vsip_scalar_f vsip_mget_f(
   const vsip_mview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_vi a3);
```

```
vsip_cscalar_f vsip_cmget_f(
   const vsip_cmview_f* a2,
   vsip_scalar_vi a2,
   vsip_scalar_vi a3);
```

Returns	Value indexed by a2, and for matrices a3. For boolean the returned value will test properly for true or false using standard ANSI C tests, but the actual value is implementation dependent.
Argument a1	Vector view from which a value will be selected and returned.
Argument a2	Index value of desired element. The first element will have an index value of 0 (zero). For matrices this is the row index.
Argument a3	This is the column index for matrices. For instance (a2,a3) = (0,0) will be the first element in the matrix, (a2,a3) = (0,1) will be the second element in the first row and (a2,a3) = (1,0) will be the first element in the second row.

getattrib

Access function to retrieve a structure containing the attributes of a view object.

```
void vsip_vgetattrib_f(
   const vsip_vview_f* a1,
   vsip_vattr_f* a2);

void vsip_cvgetattrib_f(
   const vsip_vview_f* a1,
   vsip_cvattr_f* a2);

void vsip_vgetattrib_i(
   const vsip_vview_i* a1,
   vsip_vattr_i* a2);

void vsip_vgetattrib_vi(
   const vsip_vview_vi* a1,
   vsip_vattr_vi* a2);

void vsip_vgetattrib_mi(
   const vsip_vview_mi* a1,
   vsip_vattr_mi* a2);

void vsip_vgetattrib_bl(
   const vsip_vview_vi* a1,
   vsip_vattr_bl* a2);

void vsip_mgetattrib_f(
   const vsip_mview_f* a1,
   vsip_mattr_f* a2);

void vsip_cmgetattrib_f(
   const vsip_cmview_f* a1,
   vsip_cmattr_f* a2);
```

Argument a1	Input view whose attributes will be returned.
Argument a2	Attribute structure to be filled with public attributes of input view.

getblock

Access function to retrieve the block associated with a view object.

```
vsip_block_f* vsip_vgetblock_f(
   const vsip_vview_f* a1);

vsip_cblock_f* vsip_cvgetblock_f(
   const vsip_cvview_f* a1);
```

```
vsip_block_i* vsip_vgetblock_i(
   const vsip_vview_i* a1);
vsip_block_vi* vsip_vgetblock_vi(
   const vsip_vview_vi* a1);
vsip_block_mi* vsip_vgetblock_mi(
   const vsip_vview_mi* a1);
vsip_block_bl* vsip_vgetblock_bl(
   const vsip_vview_bl* a1);
vsip_block_f* vsip_mgetblock_f(
   const vsip_mview_f* a1);
vsip_cblock_f* vsip_cmgetblock_f(
   const vsip_cmview_f* a1);
```

Returns	A block object pointer.
Argument a1	The view bound to the block object being returned.

getcollength

Access function to retrieve the column length of a matrix.

```
vsip_length vsip_mgetcollength_f(
   const vsip_mview_f* a1);
vsip_length vsip_cmgetcollength_f(
   const vsip_cmview_f* a1);
```

Returns	Number of elements in the column of a matrix view.
Argument a1	Input matrix view.

getcolstride

Access function to retrieve the column stride of a matrix.

```
vsip_length vsip_mgetcolstride_f(
   vsip_mview_f* a1);
vsip_length vsip_cmgetcolstride_f(
   vsip_cmview_f* a1);
```

Returns	Stride through the block between consecutive elements in a column.
Argument a1	Input matrix view.

getlength

Access function to retrieve the row length of a vector.

```
vsip_length vsip_vgetlength_f(
   vsip_vview_f* a1);
vsip_length vsip_cvgetrowlength_f(
   vsip_cvview_f* a1);
vsip_length vsip_vgetlength_i(
   vsip_vview_i* a1);
vsip_length vsip_vgetlength_vi(
   vsip_vview_vi* a1);
```

```
vsip_length vsip_vgetlength_mi(
   vsip_vview_mi* a1);
vsip_length vsip_vgetlength_bl(
   vsip_vview_bl* a1);
```

Returns	Number of elements in the vector view.
Argument a1	Input vector view.

getoffset

Access function to retrieve the offset from the beginning of the block associated with a view to the first element in the view. The offset is zero based and positive so that an offset of zero is the first element in the block.

```
vsip_offset vsip_vgetoffset_f(
   const vsip_vview_f* a1);
vsip_offset vsip_cvgetoffset_f(
   const vsip_cvview_f* a1);
vsip_offset vsip_vgetoffset_i(
   const vsip_vview_i* a1);
vsip_offset vsip_vgetoffset_vi(
   const vsip_vview_vi* a1);
vsip_offset vsip_vgetoffset_mi(
   const vsip_vview_mi* a1);
vsip_offset vsip_vgetoffset_bl(
   const vsip_vview_bl* a1);
vsip_offset vsip_mgetoffset_f(
   const vsip_mview_f* a1);
vsip_offset vsip_cmgetoffset_f(
   const vsip_cmview_f* a1);
```

Returns	Offset of first element of view into block bound to view.
Argument a1	Input view.

getrowlength

Access function to retrieve the row length of a matrix.

```
vsip_length vsip_mgetrowlength_f(
   const vsip_mview_f* a1);
vsip_length vsip_cmgetrowlength_f(
   const vsip_cmview_f* a1);
```

Returns	Number of elements in the row of a matrix view.
Argument a1	Input matrix view.

getrowstride

Access function to retrieve the row stride of a matrix.

```
vsip_stride vsip_mgetrowstride_f(
   const vsip_mview_f* a1);
```

```
vsip_stride vsip_cmgetrowstride_f(
   const vsip_cmview_f* a1);
```

Returns	Stride through the block between consecutive elements in a row.
Argument a1	Input matrix view.

getstride

Access function to retrieve the stride of a vector view.

```
vsip_stride vsip_vgetstride_f(
   const vsip_vview_f* a1);
```

```
vsip_stride vsip_cvgetstride_f(
   const vsip_cvview_f* a1);
```

```
vsip_stride vsip_vgetstride_i(
   const vsip_vview_i* a1);
```

```
vsip_stride vsip_vgetstride_vi(
   const vsip_vview_vi* a1);
```

```
vsip_stride vsip_vgetstride_mi(
   const vsip_vview_mi* a1);
```

```
vsip_stride vsip_vgetstride_bl(
   const vsip_vview_bl* a1);
```

Returns	Stride of vector view through it’s asscoiated block.
Argument a1	Input vector view.

hanning

Create a unit stride zero offset floating point vector and fill it with a Hanning window of chosen length.

```
vsip_vview_f* vsip_vcreate_hanning_f(
   visp_length a1,
   vsip_memory_hint hint);
```

Returns	The created vector view filled with the window coefficients.
Argument a1	The number of window coefficients.
Argument a2	Memory hint. Not supported in TASP VSIPL. recommend placing a zero here.

histo

Histogram function. This function uses a maximum value and a minimum value and the length of the output vector to calculate the bin size. Input values less than the minimum value are counted as belonging in the first element of the output vector and input values greater than the maximum value are counted as belonging in the last element of the output vector. The bin size is distributed evenly for the other elements.

```
void vsip_vhisto_f(
   const vsip_vview_f* a1,
   vsip_scalar_f* a2,
      vsip_scalar_f* a3,
      const vsip_vview_f* a4);
```

Argument a1	Input vector of values for which a histogram is desired.
Argument a2	Minimum value for which elements less than are counted in the first output element.
Argument a3	Maximum value for which elements greater than are counted in the last output element
Argument a4	Output vector of histogram counts.

hypot

Hypotenuse. Compute elementwise the square root of the sum of the squares of two input vectors.

```
void vsip_vhypot_f(
   const vsip_vview_f *a1,
   const vsip_vview_f *a2,
   const vsip_vview_f *a3);
```

Argument a1	Input vector view.
Argument a2	Input vector view.
Argument a3	Output vector view.

imag

Copy the imaginary elements of a complex vector to a real vector.

Scalar imaginary part.

```
vsip_scalar_f vsip_imag_f(
   vsip_csclar_f a1);
```

Returns	The imaginary part.
Argument a1	The input complex scalar.

Vector imaginary part.

```
void vsip_vimag_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1	Input complex vector.
Argument a2	Output vector to contain the imaginary part of the complex input vector.

imagview

Create a real view of the imaginary portion of a complex view. This is not a copy. Modifying elements in either the real view or the complex view will modify the corresponding element in the other view. Attributes of the created view are vendor dependent, and should be queried if needed.

```
vsip_vview_f* vsip_vimagview_f(
   const vsip_cvview_f* a1);
```

```
vsip_mview_f* vsip_mimagview_f(
   const vsip_cmview_f* a1);
```

Returns	View of imaginary portion of the complex view a1.
Argument a1	Complex view from which the real view of the imaginary part will be derived.

indexbool

Find an index of true values in a boolean view, placing the indexes in an vector index view.

```
vsip_length vsip_vindexbool(
   const vsip_vview_bl *a1,
   vsip_vview_vi *a2);
```

Returns	Number of true elements in argument a1.
Argument a1	Input view of boolean.
Argument a2	Output vector of indices. The indices are ordered so that the smallest index with a true value is first and the largest index with a true value is last. The length of the vector index view is modified to the number of true values, or is unmodified if no true values are found.

invclip

Using three comparison threshold values and two replacement threshold values do an elementwise clip using the following rules. Once a rule is met the condition is followed and the following rules are ignored.

$$\begin{gathered} \text{a1} < \text{a2} \Rightarrow \text{a7} = \text{a1} \\ \text{else} \\ \text{a1} < \text{a3} \Rightarrow \text{a7} = a5 \\ \text{else} \\ \text{a1} \le \text{a4} \Rightarrow \text{a7} = \text{a6} \\ \text{else} \\ \text{a7} = \text{a1} \end{gathered}$$

```
void vsip_vinvclip_f(
   const vsip_vview_f *a1,
   vsip_scalar_f a2,
   vsip_scalar_f a3,
   vsip_scalar_f a4,
   vsip_scalar_f a5,
   vsip_scalar_f a6,
   const vsip_vview_f *a7);
void vsip_vinvclip_i(
   const vsip_vview_i *a1,
   vsip_scalar_i a2,
   vsip_scalar_i a3,
   vsip_scalar_i a4,
   vsip_scalar_i a5,
      vsip_scalar_i a6,
      const vsip_vview_f *a7);
```

Argument a1	Input view.
Argument a2	Comparison threshold lower boundary.
Argument a3	Comparison threshold mid boundary.
Argument a4	Comparison threshold upper boundary.
Argument a5	Lower replacement value.
Argument a6	Upper replacement value.
Argument a7	Output view.

kaiser

Create a unit stride zero offset floating point vector and fill it with a Kaiser window of chosen length.

```
vsip_vview_f* vsip_vcreate_kaiser_f(
   visp_length a1,
   vsip_scalar_f a2,
   vsip_memory_hint a3);
```

Returns	The created vector view filled with the window coefficients.
Argument a1	The number of window coefficients.
Argument a2	Coefficient determined by user to control sidelobe levels.
Argument a3	Memory hint. Not supported in TASP VSIPL. recommend placing a zero here.

llsqsol

Linear Least Square Solver. Solve the linear least squares problem $\min\|\bar{A}\vec{x}_i - \vec{b}_i\|$ for the set of vectors $\bar{X} = [\vec{x}_o, \vec{x}_1, \ldots, \vec{x}_m]$ and $\bar{B} = [\vec{b}_o, \vec{b}_1, \ldots, \vec{b}_m]$.

```
int vsip_llsqsol_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2);
int vsip_cllsqsol_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2);
```

Returns	Zero (0) if successful, -1 if a memory allocation failure, positive if the input matrix is not of full column rank.
Argument a1	Input matrix $\bar{A}$ of size m by n. The input matrix data is overwritten in the solution process.
Argument a2	Input matrix $\bar{B}$ of size m by k, output matrix of solution $\bar{X}$. Note that the output matrix is the same VSIPL object as the input matrix. The lengths of the solutions will be n, the row length of the input matrix $\bar{A}$. It is up to the user to reset the column length of the matrix $\bar{B}$ where the solutions reside to the proper size. The solution columns will start at element zero of the input matrix column and so only a column length adjustment is required. If the input output object will be

used repeatedly it may be desirable to have a second view with the proper attributes for the output.

log

Elementwise natural (base e) logarithm of a vector.

```
void vsip_vexp_log_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

log10

Elementwise base 10 logarithm of a vector.

```
void vsip_vlog10_f*(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

logical

Functions performing logical elementwise comparisons between two input vectors.

Logical Equal. Compare two vectors elementwise for equality.

```
void vsip_vleq_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Logical greater than or equal. Compare two vectors elementwise. If the element in the first vector is greater than or equal to the element in the second input vector then a true is placed in the output vector, otherwise a false.

```
void vsip_vlge_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Logical greater than. Compare two vectors elementwise. If the element in the first vector is greater than the element in the second vector then a true is placed in the output vector, otherwise a false.

```
void vsip_vlgt_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Logical less than or equal. Compare two vectors elementwise. If the element in the first vector is less than or equal to the element in the second vector then a true is placed in the output vector, otherwise a false.

```
void vsip_vlle_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Logical less than. Compare two vectors elementwise. If the element in the first vector is less than the element in the second vector then a true is placed in the output vector, otherwise a false.

```
void vsip_vllt_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Logical not equal. Compare two vectors elementwise. If the element in the first vector is not equal to the element in the second vector then a true is placed in the output vector, otherwise a false.

```
void vsip_vlle_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_bl* a3);
```

Argument a1	Input vector.
Argument a2	Input vector.
Argument a3	Output boolean vector of true and false values.

lud

Matrix lower upper decomposition. Solve linear systems using a Gaussian decomposition of a square matrix.

Create an LUD object.

```
vsip_lu_f* vsip_lud_create_f(
   vsip_length a1);
```

```
vsip_clu_f* vsip_clud_create_f(
   vsip_length a1);
```

Returns	The LUD object.
Argument a1	Number of rows (and columns) in the expected matrix decomposition.

Compute the LUD decomposition. The matrix decomposed is overwritten and bound to the LUD object. This matrix must not be modified as long as the LUD object is needed. The lud object is created using the create function above.

```
int vsip_lud_f(
   vsip_lu_f* a1,
   const vsip_mview_f* a2);
```

```
int vsip_clud_f(
   vsip_clu_f* a1,
   const vsip_cmview_f* a2);
```

Returns	Zero (0) on success.
Argument a1	Input/Output LUD object.
Argument a2	Input matrix to be decomposed. The matrix is overwritten by the decomposition.

Solve a square linear system.

$$\mathrm{op}(\bar{A})\vec{x}_i = \vec{b}$$

$$\bar{X} = [\vec{x}_o, \vec{x}_1, \ldots, \vec{x}_m]$$

$$\bar{B} = [\vec{b}_o, \vec{b}_1, \ldots, \vec{b}_m]$$

```
int vsip_lusol_f(
   const vsip_lu_f* a1,
   vsip_mat_op a2,
   const vsip_mview_f* a3);
```

```
int vsip_clusol_f(
   const vsip_lu_f* a1,
   vsip_mat_op a2,
   const vsip_mview_f* a3);
```

Returns	Zero (0) on success.
Argument a1	Input LUD object which has been created using **`vsip_lud_create_f`** and a decomposition matrix computed using **`vsip_lud_f.`** The LUD object contains the decomposed matrix $\bar{A}$.
Argument a2	Matrix Operator flag. This flag operates on the matrix $\bar{A}$.
Argument a3	Input/Output matrix of Vectors to solve for so that $\bar{B}$ is a3 on input, and $\bar{X}$ is a3 on output.

Destroy LUD object.

```
int vsip_lud_destroy_f(
   vsip_lu_f* a1);
```

```
int vsip_clud_destroy_f(
   vsip_clu_f* a1);
```

Returns	Zero on success.
Argument a1	LUD object to be destroyed. The matrix A associated with the LUD object is not destroyed here, and must be destroyed using the matrix destroy functions.

Get LUD attributes. The only public attribute is the matrix size N.

```
void vsip_lud_gatattr_f(
   const vsip_lu_f* a1,
   vsip_lu_attr_f* a2);
```

```
void vsip_clud_getattr_f(
   const vsip_clu_f* a1,
   vsip_clu_attr_f* a2);
```

Argument a1	LUD object.
Argument a2	LUD attribute object.

mag

Magnitude.

Scalar Magnitude.

```
vsip_scalar_f vsip_cmag_f(
   vsip_cscalar_f *a1);
```

Returns	The magnitude of the input.
Argument a1	Input scalar value.

Elementwise find the magnitude of a vectors elements and place them in an output vector.

```
void vsip_vmag_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvmag_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_vmag_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2);
```

Argument a1	Input vector.
Argument a2	Output vector of magnitudes.

matindex

Scalar function to create a scalar matrix index.

```
vsip_scalar_mi vsip_matindex(
   vsip_scalar_vi a1,
   vsip_scalar_vi a2);
```

```
void vsip_MATINDEX(
   vsip_scalar_vi a1,
   vsip_scalar_vi a2,
   vsip_scalar_mi *a3);
```

Returns	For non-void the scalar matrix index value.
Argument a1	Row index element for matrix index.

Argument a2 Column index element for matrix index.

Argument a3 For void scalar matrix index the scalar matrix index $(a1, a2)$

Scalar functions to extract row index and column index from scalar matrix index.

```
vsip_scalar_vi vsip_rowindex(
   vsip_scalar_mi a1);
```

```
vsip_scalar_vi vsip_colindex(
   vsip_scalar_mi a1);
```

Returns Extracted row or column index.

Argument a1 Input matrix index.

max

Compare two vectors element by element and place the maximum value of each element comparison in an output vector.

```
void vsip_vmax_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector.

Argument a2 Input vector.

Argument a3 Output vector.

maxmg

Maximum magnitude selection. Compare two real vectors elementwise and output the maximum magnitude into the output vector.

```
void vsip_vmaxmg_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector.

Argument a2 Input vector.

Argument a3 Output vector.

maxmgval

Find the maximum magnitude among all the values of a real vector.

```
vsip_scalar_f vsip_vmaxmgval_f(
   const vsip_vview_f* a1,
   vsip_index* a2);
```

Returns Maximum magnitude value of the vector.

Argument a1 Input vector.

Argument a2	Pointer to index (in input vector) of location of the maximum magnitude. If more than one element has the maximum magnitude value then the index of the first element is returned. If the pointer is a null value the index is ignored.

maxval

Find the maximum value of a vector and return it and it’s index.

```
vsip_scalar_f vsip_vmaxval_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi* a2);
```

Returns	Maximum value of the input vector
Argument a1	Input Vector.
Argument a2	If a2 is not a null value the index of the (first) maximum value is returned.

meansqval

Find the average of the magnitude squared elements of a view.

```
vsip_scalar_f vsip_vmeansqval_f(
   const vsip_vview_f* a1);
vsip_scalar_f vsip_cvmeansqval_f(
   const vsip_cvview_f* a1);
```

Returns	The mean value of the magnitude squared values of the elements of the input.
Argument a1	The input view.

meanval

Find the average of the elements of a view.

```
vsip_scalar_f vsip_vmeanval_f(
   const vsip_vview_f* a1);
vsip_cscalar_f vsip_cvmeanval_f(
   const vsip_cvview_f* a1);
```

Returns	The mean value of the elements of the view.
Argument a1	Input view.

mherm

Matrix Hermitian. Do a conjugate transpose of a complex matrix. May be done in place only if the input matrix is square.

```
void vsip_cmherm_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2);
```

Argument a1	Input matrix view.
Argument a2	Output matrix view.

min

Compare two vectors element by element and place the minimum value of each element comparison in an output vector.

```
void vsip_vmin_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1	Input vector
Argument a2	Input vector
Argument a3	Output vector

minmg

Minimum magnitude selection. Compare two real vectors elementwise and output the minimum magnitude into the output vector.

```
void vsip_vminmg_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1	Input vector.
Argument a2	Input vector.
Argument a3	Output vector.

minmgval

Find the minimum magnitude among all the values of a real vector.

```
vsip_scalar_f vsip_vminmgval_f(
   const vsip_vview_f* a1,
   vsip_index* a2);
```

Argument a1	Input vector.
Argument a2	If not null, the index of the minimum magnitude is returned. If more than one element equals the minimum magnitude then the index of the first equal element is returned.

minval

Find the minimum value of a vector and return it and it’s index.

```
vsip_scalar_f vsip_vminval_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi* a2);
```

Returns	Minimum value of the input vector.
Argument a1	Input vector.
Argument a2	If a2 is not a null value then the index of the minimum value is returned. If more than one element equals the minimum value then the index of the first minimum value is returned

modulate

Modulate a vector by a specified frequency. Phase and information is passed to, and returned from the modulate function to allow continuous modulation. The modulation formula is

$$a4 = (\cos[t(a2) + a3] + j\sin[t(a2) + a3])a1$$

where t is the element index and j is the complex imaginary multiplier.

```
vsip_scalar_f vsip_vmodulate_f(
   const vsip_vview_f* a1,
   vsip_scalar_f a2,
   vsip_scalar_f a3,
   const vsip_cvview_f* a3);
```

```
vsip_scalar_f vsip_vmodulate_f(
   const vsip_cvview_f* a1,
   vsip_scalar_f a2,
   vsip_scalar_f a3,
   const vsip_cvview_f* a3);
```

Returns	Next phase value.
Argument a1	Input vector view.
Argument a2	Input frequency value.
Argument a3	Input phase value.
Argument a4	Output complex vector view.

mprod

Matrix Products. These are standard linear algebra products of matrix views with vector or matrix views. Sizes of input and output views must match the standard linear algebra definitions.

Matrix Product

```
void vsip_mprod_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2,
   const vsip_mview_f* a3);
```

```
void vsip_cmprod_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cmview_f* a3);
```

Argument a1	First input matrix.
Argument a2	Second input matrix.
Argument a3	Output matrix

Conjugate matrix product. Matrix multiply the first input matrix times the complex conjugate of the second input matrix.

```
void vsip_cmprodj_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cmview_f* a3);
```

Argument a1 First input matrix.

Argument a2 Second input matrix.

Argument a3 Output matrix

Hermitian matrix product. Matrix multiply the first input matrix times the complex conjugate transpose of the second input matrix.

```
void vsip_cmprodj_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cmview_f* a3);
```

Argument a1 First input matrix.

Argument a2 Second input matrix.

Argument a3 Output matrix

Vector matrix product.

```
void vsip_vmprod_f(
   const vsip_vview_f* a1,
   const vsip_mview_f* a2,
   const vsip_vview_f* a3);
```

```
void vsip_cvmprod_f(
   const vsip_cvview_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Input vector view. The length of the vector view must match the number of rows (column length) of argument a2.

Argument a2 Input matrix view.

Argument a3 Output vector view.

Matrix vector product.

```
void vsip_mvprod_f(
   const vsip_mview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

```
void vsip_cmvprod_f(
   const vsip_cmview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input matrix view.

Argument a2 The length of the vector view must match the number of columns (row length) of argument a1.

Argument a3 Output vector view.

Matrix transpose product. Matrix product of the first input matrix with the transpose of the second input matrix.

```
void vsip_mprodt_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2,
   const vsip_mview_f* a3);
void vsip_cmprodt_f(
   const vsip_cmview_f* a1,
   const vsip_cmview_f* a2,
   const vsip_cmview_f* a3);
```

Argument a1 First input matrix of size MxP .

Argument a2 Second input matrix of size NxP

Argument a3 Output matrix of size MxN

mtrans

Matrix transpose. May be done in place only if the matrix is square.

```
void vsip_mtrans_f(
   const vsip_mview_f* a1,
   const vsip_mview_f* a2);
void vsip_cmtrans_f(
   const vsip_cmview_f* a1,
   const vsip_mview_f* a2);
```

Argument a1 Input matrix view.

Argument a2 Output matrix view

mul

Multiply two objects element by element.

Multiply two scalars.

```
vsip_cscalar_f vsip_cmul_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2);
vsip_cscalar_f vsip_rcmul_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2);
void vsip_CMUL_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f* a3);
void vsip_RCMUL_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f *a3);
```

Returns	For non void scalar functions the product of the arguments.
Argument a1	Input scalar
Argument a2	Input scalar
Argument a3	Pointer to output complex scalar.

Multiply two vectors elementwise.

```
void vsip_vmul_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
void vsip_vmul_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
void vsip_cvmul_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
void vsip_rcvmul_f(
   const vsip_vview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1	Input vector view.
Argument a2	Input vector view.
Argument a3	Output vector view.

Multiply a scalar times a vector elementwise.

```
void vsip_svmul_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
void vsip_svmul_i(
   vsip_scalar_i a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
void vsip_csvmul_f(
   vsip_cscalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
void vsip_rscvmul_f(
   vsip_scalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1	Input scalar.
Argument a2	Input vector view.
Argument a3	Output vector view.

Complex conjugate multiply.

Scalar conjugate multiply. Multiply the first input scalar times the conjugate of the second input scalar

```
vsip_cscalar_f vsip_cjmul_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2);
```

```
void vsip_CJMUL_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f* a3);
```

`Returns`	For non void the product of the arguments.
`Argument a1`	First input scalar.
`Argument a2`	Second input scalar
`Argument a3`	Pointer to output complex scalar.

Elementwise multiply the first input times the conjugate of the second input.

```
void vsip_cvjmul_f(
   vsip_cvview_f* a1,
   vsip_cvview_f* a2,
   vsip_cvview_f* a3);
```

`Argument a1`	The first input vector view.
`Argument a2`	The second input vector view.
`Argument a3`	The output vector.

Vector Matrix elementwise multiply. Elementwise multiply the elements of a vector times the elements of the rows or columns of a matrix. The length of the vector must be the same length as the selected row or column direction.

```
void vsip_vmmul_f(
   const vsip_vview_f *a1,
   const vsip_mview_f *a2,
   vsip_major major,
   const vsip_mview_f *a3);
```

```
void vsip_cvmmul_f(
   const vsip_cvview_f *a1,
   const vsip_cmview_f *a2,
   vsip_major major,
   const vsip_cmview_f *a3);
```

```
void vsip_rvcmmul_f(
   const vsip_vview_f *a1,
   const vsip_cmview_f *a2,
   vsip_major major,
   const vsip_cmview_f *a3);
```

`Argument a1`	Input vector view.
`Argument a2`	Input matrix view.

Argument a3 Output matrix view.

neg

Perform an unary minus.

Scalar unary minus

```
vsip_cscalar_f vsip_cneg_f(
   vsip_cscalar_f a1);
```

```
void vsip_CNEG_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f* a1);
```

Returns For non void the negative of the argument.

Argument a1 Input scalar.

Argument a2 Pointer to output scalar.

Elementwise perform an unary minus on a view.

```
void vsip_vneg_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvneg_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

```
void vsip_vneg_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2);
```

Argument a1 Input vector

Argument a2 Output vector

not

Elementwise calculate the bitwise “NOT” for an integer view, or a logical “NOT” for an boolean view.

```
void vsip_vnot_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2);
```

```
void vsip_vnot_bl(
   const vsip_vview_bl* a1,
   const vsip_vview_bl* a2);
```

Argument a1 Input view.

Argument a2 Output view.

or

Performs a bitwise “OR” operation between two integer views, or a logical “OR” between two boolean views.

```
void vsip_vor_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
```

```
void vsip_vor_bl(
   const vsip_vview_bl* a1,
   const vsip_vview_bl* a2,
   const vsip_vview_bl* a3);
```

Argument a1 Input view.

Argument a2 Input view.

Argument a3 Output view.

outer

Compute a scaled outer product of two vectors. $\bar{C} = \alpha \vec{x} \vec{y}^T$

```
void vsip_vouter_f(
   const vsip_scalar_f a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_mview_f* a4);
```

```
void vsip_cvouter_f(
   const vsip_cscalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cmview_f* a4);
```

Argument a1 Scaling factor α.

Argument a2 Input vector $\vec{x}$.

Argument a3 Input vector $\vec{y}$

Argument a4 Output matrix $\bar{C}$

polar

Convert complex rectangular to polar notation. All VSIPL complex numbers are in rectangular notation. Conversion to polar requires output into two real objects.

Scalar functionality.

```
void vsip_polar_f(
   vsip_cscalar_f a1,
   vsip_scalar_f a2,
   vsip_scalar_f a3);
```

Vector functionality.

```
void vsip_vpolar_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

`Argument a1` Input complex.

`Argument a2` Output radius (square root of sum of squares of real and imaginary of input).

`Argument a3` Output angle (atan2(real a1, imaginary a1)).

put

Put an element into a vector.

```
void vsip_vput_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_f a4);
void vsip_cvput_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2,
   vsip_cscalar_f a4);
void vsip_vput_i(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_i a4);
void vsip_vput_vi(
   const vsip_vview_vi* a1,
   vsip_scalar_vi a2,
   vsip_scalar_vi a4);
void vsip_vput_mi(
   const vsip_vview_mi* a1,
   vsip_scalar_vi a2,
   vsip_scalar_mi a4);
void vsip_vput_bl(
   const vsip_vview_bl* a1,
   vsip_scalar_vi a2,
   vsip_scalar_bl a4);
void vsip_mput_f(
   const vsip_mview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_vi a3
   vsip_scalar_f a4);
void vsip_cmput_f(
   const vsip_cmview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_vi a3,
   vsip_cscalar_f a4);
```

`Argument a1` View into which an element will be placed.

`Argument a2` Index value of desired element location. The first element in the view will have an index value of 0 (zero). For matrix puts this is the row index.

Argument a3	Column index value for matrix put. The first element is 0.
Argument a4	Value to place in view.

putattrib

Access function to set the attributes of a view object using the public attributes object.

```
vsip_vview_f* vsip_vputattrib_f(
   vsip_vview_f* a1,
   const vsip_vattr_f* a2);
```

```
vsip_cvview_f* vsip_cvputattrib_f(
   vsip_cvview_f* a1,
   const vsip_cvattr_f* a2);
```

```
vsip_vview_i* vsip_vputattrib_i(
   vsip_vview_i* a1,
   const vsip_vattr_i* a2);
```

```
vsip_vview_vi* vsip_vputattrib_vi(
   vsip_vview_vi* a1,
   const vsip_vattr_vi* a2);
```

```
vsip_vview_mi* vsip_vputattrib_mi(
   vsip_vview_mi* a1,
   const vsip_vattr_mi* a2);
```

```
vsip_vview_bl* vsip_vputattrib_bl(
   vsip_vview_bl* a1,
   const vsip_vattr_bl* a2);
```

```
vsip_mview_f* vsip_mputattrib_f(
   vsip_mview_f* a1,
   const vsip_mattr_f* a2);
```

```
vsip_cmview_f* vsip_cmputattrib_f(
   vsip_cmview_f* a1,
   const vsip_cmattr_f* a2);
```

Returns	Pointer to input view as a convenience.
Argument a1	Input vector whose attributes will be modified.
Argument a2	Attribute structure to be filled with attributes of input vector. Note that the block is ignored when putting an attribute.

putcollength

Replace the column length in a matrix view.

```
vsip_mview_f* vsip_mputcollength_f(
   vsip_mview_f* a1,
   vsip_length a3);
```

```
vsip_cmview_f* vsip_cmputcollength_f(
   vsip_cmview_f* a1,
   vsip_length a3);
```

Returns	Pointer to input view as a convenience.
Argument a1	Input matrix whose column length will be modified.

Argument a2 New length.

putcolstride

Replace the column stride in a matrix view.

```
vsip_mview_f* vsip_mputcolstride_f(
   vsip_mview_f* a1,
   vsip_stride a3);
```

```
vsip_cmview_f* vsip_cmputcolstride_f(
   vsip_cmview_f* a1,
   vsip_stride a3);
```

Returns	Pointer to input view as a convenience.
Argument a1	Input matrix whose column stride will be modified.
Argument a2	New stride.

putoffset

Access function to set the offset of a view object

```
vsip_vview_f* vsip_vputoffset_f(
   vsip_vview_f* a1,
   vsip_offset a2);
```

```
vsip_cvview_f* vsip_cvputoffset_f(
   vsip_cvview_f* a1,
   vsip_offset a2);
```

```
vsip_vview_i* vsip_vputoffset_i(
   vsip_vview_i* a1,
   vsip_offset a2);
```

```
vsip_vview_vi* vsip_vputoffset_vi(
   vsip_vview_vi* a1,
   vsip_offset a2);
```

```
vsip_vview_mi* vsip_vputoffset_mi(
   vsip_vview_mi* a1,
   vsip_offset a2);
```

```
vsip_vview_bl* vsip_vputoffset_bl(
   vsip_vview_bl* a1,
   vsip_offset a2);
```

```
vsip_mview_f* vsip_mputoffset_f(
   vsip_mview_f* a1,
   vsip_offset a2);
```

```
vsip_cmview_f* vsip_cmputoffset_f(
   vsip_cmview_f* a1,
   vsip_offset a2);
```

Returns	Pointer to input view as a convenience.
Argument a1	View whose offset is to be reset.

Argument a2	Offset value. An offset of 0 (zero) will place the offset at the first element of the block.

putrowlength

Replace the row length in a matrix view.

```
vsip_mview_f* vsip_mputrowlength_f(
   vsip_mview_f* a1,
   vsip_length a3);
```

```
vsip_cmview_f* vsip_cmputrowlength_f(
   vsip_cmview_f* a1,
   vsip_length a3);
```

Returns	Pointer to input view as a convenience.
Argument a1	Input matrix whose row length will be modified.
Argument a2	New length.

putrowstride

Replace the row stride in a matrix view.

```
vsip_mview_f* vsip_mputrowstride_f(
   vsip_mview_f* a1,
   vsip_stride a3);
```

```
vsip_cmview_f* vsip_cmputrowstride_f(
   vsip_cmview_f* a1,
   vsip_stride a3);
```

Returns	Pointer to input view as a convenience.
Argument a1	Input matrix whose row stride will be modified.
Argument a2	New stride.

putstride

Access function to set the stride of a vector view object.

```
vsip_vview_f* vsip_vputstride_f(
   vsip_vview_f* a1,
   vsip_stride a2);
```

```
vsip_cvview_f* vsip_cvputstride_f(
   vsip_cvview_f* a1,
   vsip_stride a2);
```

```
vsip_vview_i* vsip_vputstride_i(
   vsip_vview_i* a1,
   vsip_stride a2);
```

```
vsip_vview_vi* vsip_vputstride_vi(
   vsip_vview_vi* a1,
   vsip_stride a2);
```

```
vsip_vview_mi* vsip_vputstride_mi(
   vsip_vview_mi* a1,
   vsip_stride a2);
```

```
vsip_vview_bl* vsip_vputstride_bl(
   vsip_vview_bl* a1,
   vsip_stride a2);
```

Returns	Pointer to input view as a convenience.
Argument a1	Vector view whose stride is to be reset.
Argument a2	Stride value. Strides may be positive, negative or zero.

putlength

Access function to set the length of a vector view object

```
vsip_vview_f* vsip_vputlength_f(
   vsip_vview_f* a1,
   vsip_length a2);
```

```
vsip_cvview_f* vsip_cvputlength_f(
   vsip_cvview_f* a1,
   vsip_length a2);
```

```
vsip_vview_i* vsip_vputlength_i(
   vsip_vview_i* a1,
   vsip_length a2);
```

```
vsip_vview_vi* vsip_vputlength_vi(
   vsip_vview_vi* a1,
   vsip_length a2);
```

```
vsip_vview_mi* vsip_vputlength_mi(
   vsip_vview_mi* a1,
   vsip_length a2);
```

```
vsip_vview_bl* vsip_vputlength_bl(
   vsip_vview_bl* a1,
   vsip_length a2);
```

Returns	Pointer to input view as a convenience.
Argument a1	Vector whose length is to be reset.
Argument a2	Length value.

qrd

Matrix decomposition using the QR method. This function set is used for solving linear systems, in particular over determined systems.

QRD create function. Create the QRD object.

```
vsip_qr_f* vsip_qrd_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_qrd_qopt a3);
```

```
vsip_cqr_f* vsip_cqrd_create_f(
   vsip_length a1,
   vsip_length a2,
   vsip_qrd_qopt a3);
```

Returns A QRD object.

Argument a1 Number of rows in the input matrix $\overline{A}$ in the QRD decomposition function **vsip_qrd_f** or **vsip_cqrd_f**.

Argument a2 Number of columns in the input matrix $\overline{A}$ in the QRD decomposition function **vsip_qrd_f** or **vsip_cqrd_f**.

Argument a3 Enumerated type definition indicating what type of QRD information is required in the matrix decomposition, either no $\overline{Q}$, full $\overline{Q}$, or skinny $\overline{Q}$.

Decompose the input matrix $\overline{A}$ and bind the decomposition to the QRD object. Note that the $\overline{A}$ matrix is used in the decomposition, and should not be modified or destroyed until the QRD object is no longer needed. For the following the size of $\overline{A}$ is m by n.

```
int vsip_qrd_f(
   vsip_qr_f* a1,
   const vsip_mview_f* a2);
int vsip_cqrd_f(
   vsip_cqr_f* a1,
   const vsip_cmview_f* a2);
```

Returns Zero (0) on success.

Argument a1 Input QRD object which will contain the $\overline{A}$ decomposition information.

Argument a2 Input matrix $\overline{A}$ to be decomposed using QR methods.

Using the QRD object calculate the product of the matrix $\overline{Q}$ from the QR decomposition of matrix $\overline{A}$ where $\overline{A}$ is of size m_a by n_a and $m_a \geq n_a$. To use this function the QRD object must have been created with one of the save Q options. If **VSIP_QRD_SAVEQ** was specified the size of the implied $\overline{Q}$ matrix is m_a by m_a. if **VSIP_QRD_SAVEQ1** was specified the size of the implied $\overline{Q}$ matrix is m_a by n_a.

This function preforms the operation.

$$\text{op}(\overline{Q}) \cdot \overline{C} \text{ or } \overline{C} \cdot \text{op}(\overline{Q})$$

Note that this matrix product is done in place and is of the form $\overline{H} \cdot \overline{K} = \overline{C}$ where $\overline{H}$ is of size m by n, $\overline{K}$ is of size n by k and $\overline{C}$ is of size m by k. Either matrix

$\bar{H}$ or $\bar{K}$ (depending upon the option selected) may be required to be in place with $\bar{C}$ since the $\mathrm{op}(\bar{Q})$ may be either on the left or right. The output matrix may be the same size as the input, or it may be larger or smaller than the input matrix, depending on the input sizes of $\bar{H}$ and $\bar{K}$, and which input matrix represents the output matrix. The following in-place rules are followed. (1) The elements of the input/output matrices are arranged in there natural matrix element locations in the upper left corner of the input/output matrix *view*. (2) The input/output matrix *view* will be of the input. (3) If the output is larger than the input then the strides of the input matrix must be sufficient so that the output data may be contained in the block.

```
int vsip_qrdprodq_f(
   const vsip_qr_f* a1,
   vsip_mat_op a2,
   vsip_mat_side a3,
   const vsip_mview_f* a4);
```

```
int vsip_cqrdprodq_f(
   const vsip_cqr_f* a1,
   vsip_mat_op a2,
   vsip_mat_side a3,
   const vsip_cmview_f* a4);
```

Returns	Zero (0) on success.
Argument a1	Input QRD object.
Argument a2	Input Operator flag. For real case only the no transpose and transpose cases are supported. For the complex case only the no transpose and hermitian transpose case are supported.
Argument a3	Input flag to cause the matrix multiply to happen on the left or the right.
Argument a4	Input matrix to be multiplied with $\mathrm{op}(\bar{Q})$, and the output matrix of result. The attributes of the input and output matrix are the same, and the row and column lengths may need to be adjusted to fit the output matrix data space. Note that if the input and output matrix are not the same size it may be convenient to define a second view of a4 which describes the input or the output, whichever is smaller.

Solve linear system using QRD object. The solution is based on the $\bar{R}$ matrix from the QR decomposition, and the linear system solved is of the form $\mathrm{op}(\bar{R})\vec{x}_i = \alpha\vec{b}_i$. The solution is done (in place) for a set of input vectors $\bar{B} = [\vec{b}_0, \vec{b}_1, \ldots, \vec{b}_k]$ and output vectors $\bar{X} = [\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_k]$.

```
int vsip_qrdsolr_f(
   const vsip_qr_f* a1,
   vsip_mat_op a2,
   vsip_scalar_f a3,
   const vsip_mview_f* a4);
```

```
int vsip_cqrdsolr_f(
   const vsip_cqr_f* a1,
   vsip_mat_op a2,
   vsip_scalar_f a4,
   const vsip_cmview_f* a4);
```

Returns Zero (0) on success.

Argument a1 Input QRD object.

Argument a2 Input operator flag to determine form of $\bar{R}$.

Argument a3 Input scale factor.

Argument a4 Input matrix $\bar{B}$ and output matrix $\bar{X}$.

Solve a covariance or linear least square problem. The covariance problem solved is of the form $\bar{A}^T\bar{A}\vec{x}_i = \vec{b}_i$. The solution is done (in place) for a set of input vectors $\bar{B} = [\vec{b}_0, \vec{b}_1, \ldots, \vec{b}_k]$ and output vectors $\bar{X} = [\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_k]$.

```
int vsip_qrsol_f(
   vsip_qr_f* a1,
   vsip_qrd_prob a2,
   const vsip_mview_f *a3);
int vsip_cqrsol_f(
   vsip_cqr_f* a1,
   vsip_qrd_prob a2,
   const vsip_cmview_f* a3);
```

Returns Zero (0) on success.

Argument a1 Input QRD object.

Argument a2 Flag to determine if the least squares problem is solved, or the covariance problem is solved.vsip_mrealview_f

Argument a3 Input view $\bar{B}$ to solve for, and output view $\bar{X}$. Note that for the covariance problem $\bar{B}$ and $\bar{X}$ are of the same size, but for the least squares problem $\bar{B}$ has a column length greater than or equal to $\bar{X}$. The in-place rule is that the output matrix goes in the upper left corner of the input matrix in natural order, and the view attributes are unchanged for input and output. It may be convenient to define a second view of a3 defining the output.

Get the attributes of a QRD object.

```
void vsip_qrd_getattr_f(
   const vsip_qr_f *a1,
   vsip_qrd_attr_f *a2);
void vsip_cqrd_getattr_f(
   const vsip_cqr_f *a1,
   vsip_cqrd_attr_f *a2);
```

Argument a1 QRD object whose attributes are being queried.

Argument a2 QRD attribute object.

Destroy a QRD object.

```
int vsip_qrd_destroy_f(
   vsip_qr_f *a1);
```

```
int vsip_cqrd_destroy_f(
   vsip_cqr_f *a2);
```

Argument a1	QRD object to be destroyed.

ramp

Fill a vector with an initial value plus some increment times the vector index.

```
void vsip_vramp_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2,
   const vsip_vview_f* a3);
```

```
void vsip_vramp_i(
   vsip_scalar_i a1,
   vsip_scalar_i a2,
   const vsip_vview_i* a3);
```

Argument a1	Starting value of ramp.
Argument a2	Ramp increment value.
Argument a3	Output vector containing ramp values.

rand

Generate a uniform random sequence. The current non-portable sequence in the TASP VSIPL implementation is based on the congruential sequence
$X_n = [(1664525)X_{n-1} + 1013904223]\mathrm{mod}(2^{32})$.
The number produced is normalized to a float value between zero and one.

Create a random state object.

```
vsip_randstate* vsip_randcreate(
   vsip_index a1,
   vsip_index a2,
   vsip_index a3,
   vsip_rng a4);
```

Returns	A random state object
Argument a1	The initial seed value for the random state object.
Argument a2	The total number of independent random state objects needed.
Argument a3	The particular random state object needed out of the number specified in argument a2. Numbering starts at 1. For instance if a2 is 3 then a3 will be 1, 2, or 3.
Argument a4	A flag to indicate whether the desired generator is the portable generator defined in the VSIPL specification, or a non-portable generator which is implementation dependent. The non-portable generator may be the same as the portable generator (For TASP VSIPL they are not the same).

Create the next random number from random state object.

Scalar function for uniform random number generator. Generates uniformly distributed floats bounded by zero and one.

```
vsip_scalar_f vsip_randu_f(
   vsip_randstate *a1);
```

```
vsip_cscalar_f vsip_crandu_f(
   vsip_randstate *a1);
```

Scalar function for $N(0, 1)$ Gaussian random number generator.

```
vsip_scalar_f vsip_randn_f(
   vsip_randstate *a1);
```

```
vsip_cscalar_f vsip_crandn_f(
   vsip_randstate *a1);
```

`Returns`	A random number
`Argument a1`	Random state object, created by randstate.

Create a vector of random numbers from the random state object.

Vector function for uniform random number generator. Generates uniformly distributed floats bounded by zero and one.

```
void vsip_vrandu_f(
   vsip_randstate *a1,
   const vsip_vview_f *a2);
```

```
void vsip_cvrandu_f(
   vsip_randstate *a1,
   const vsip_cvview_f *a2);
```

Vector function for $N(0, 1)$ Gaussian random number generator.

```
void vsip_vrandn_f(
   vsip_randstate *a1,
   const vsip_vview_f *a2);
```

```
void vsip_cvrandn_f(
   vsip_randstate *a1,
   const vsip_cvview_f *a2);
```

`Argument a1`	The random state operator
`Argument a2`	A vector view to be filled with sequential numbers from the generator specified by the randstate object.

Destroy the random state object

```
int vsip_randdestroy(
   vsip_randstate *a1);
```

`Returns`	0 on success.
`Argument a1`	The random state object to be destroyed.

real

Copy the imaginary elements of a complex vector to a real vector.

Scalar real part.

```
vsip_scalar_f vsip_real_f(
   vsip_csclar_f a1);
```

Returns	The real part.
Argument a1	The input complex scalar.

Vector real part.

```
void vsip_vreal_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1	Input complex vector.
Argument a2	Output vector to contain real part of input vector.

realview

Create a real view of the real portion of a complex view. This is not a copy. Modifying elements in either the real view or the complex view will modify the corresponding element in the other view.

```
vsip_vview_f* vsip_vrealview_f(
   const vsip_cvview_f* a1);
```

```
vsip_mview_f* vsip_mrealview_f(
   const vsip_mview_f* a1);
```

Returns	Vector view of the real portion of the complex view a1.
Argument a1	Complex vector view from which the real view of the real part will be derived.

recip

Find the reciprocal value.

Scalar Reciprocal

```
vsip_cscalar_f vsip_crecip_f(
   vsip_cscalar_f a1);
```

```
void vsip_CRECIP_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f *a2);
```

Returns	For non void scalar the reciprocal of the argument.
Argument a1	Input scalar.
Argument a2	Output of the reciprical of the input for non-void scalar function.

Elementwise find the reciprocal of a vectors elements and place them in an output vector.

```
void vsip_vrecip_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvrecip_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

`Argument a1`	Input vector
`Argument a2`	Output vector

`rect`

Convert rectangular notation to complex rectangular. VSIPL has no polar scalar storage. Polar storage is in two real objects. Complex storage is always in rectangular form.

Scalar rectangular. Convert two real scalars representing a number in polar notation to a complex number in rectangular notation.

```
vsip_cscalar_f vsip_rect_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2);
```

```
void vsip_RECT_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2,
   vsip_cscalar_f *a3);
```

`Returns`	The rectangular notation complex scalar.
`Argument a1`	Magnitude (radius) of polar notation scalar.
`Argument a2`	Angle of radius vector of polar notation scalar.

Vector rectangular. Elementwise convert two real vectors representing (pairwise) numbers in polar notation to a complex vector in rectangular notation.

```
void vsip_vrect_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2,
   vsip_cvview_f* a3);
```

`Argument a1`	Input vector representing magnitude of polar notation.
`Argument a2`	Input vector representing angle of polar notation.
`Argument a3`	Output complex vector in complex rectangular notation.

`rowview`

Create a vector view of a selected row of a matrix

```
vsip_vview_f* vsip_mrowview_f(
   const vsip_mview_f* a1,
   vsip_index a2);
```

```
vsip_cvview_f* vsip_cmrowview_f(
   const vsip_cmview_f* a1,
   vsip_index a2);
```

Returns	A vector view of the selected row, or a NULL if the memory allocation for the view object fails.
Argument a1	Input view.
Argument a2	Index of desired view. Indices are zero based so that the first (top) row of the matrix has index zero.

rsqrt

Reciprocal square root. Find the reciprocal square root of the elements of a view.

```
void vsip_vrsqrt_f(
   const vsip_vview_f *a1,
   const vwip_vview_f *a2);
```

Argument a1	Input view.
Argument a2	Output view.

scatter

An index vector and an input vector (of the same length) are indexed elementwise. The input vector value is placed in an output view based on the index vector value. The only requirement on the output view is that the index vector values are valid indices into the output view. For the core profile only vector views are supported for the output.

```
void vsip_vscatter_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_vi* a3);
void vsip_cvscatter_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_vi* a3);
void vsip_vscatter_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_vi* a3);
```

Argument a1	Input vector view.
Argument a2	Output view.
Argument a3	Input vector view of indices (index vector).

sin

Elementwise Sine of a vector

```
void vsip_vsin_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector in radian format.

Argument a2 output vector of Sine values.

sq

Elementwise find the square of a vectors elements

```
void vsip_vsq_f(
   const vsip_vview_f* a1,
   cons vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector of element squares from the input vector.

sqrt

Square Root

Scalar Square Root

```
vsip_cscalar_f vsip_csqrt_f(
   vsip_cscalar_f a1);
```

```
void vsip_CSQRT_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f *a2);
```

Returns For non-void scalar the square root of the input.

Argument a1 Input value.

Argument a2 For void scalar the square root of the input.

Elementwise square root of a vector.

```
void vsip_vsqrt_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvsqrt_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

sub

Subtract the second input from the first input.

Scalar Subtraction

```
vsip_cscalar_f vsip_csub_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2);
```

```
vsip_cscalar_f vsip_rcsub_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2);
```

```
vsip_cscalar_f vsip_crsub_f(
   vsip_cscalar_f a1,
   vsip_scalar_f a2);
```

```
void vsip_CSUB_f(
   vsip_cscalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f *a3);
```

```
void vsip_RCSUB_f(
   vsip_scalar_f a1,
   vsip_cscalar_f a2,
   vsip_cscalar_f *a3);
```

```
void vsip_CRSUB_f(
   vsip_cscalar_f a1,
   vsip_scalar_f a2,
   vsip_cscalar_f *a3);
```

Returns	For non-void scalars the result of the second input subtracted from the first input.
Argument a1	First input.
Argument a2	Second input
Argument a3	For void scalars the result of the second input subtracted from the first input.

Subtract a vector from a scalar

```
void vsip_svsub_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

```
void vsip_svsub_i(
   vsip_scalar_i a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
```

```
void vsip_csvsub_f(
   vsip_cscalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

```
void vsip_rscvsub_f(
   vsip_scalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1	Input scalar.
Argument a2	Input vector.
Argument a3	Output vector.

Subtract two vectors element by element

```
void vsip_vsub_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

```
void vsip_vsub_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
```

```
void vsip_cvsub_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

```
void vsip_rcvsub_f(
   const vsip_vview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

```
void vsip_crvsub_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1	First input argument
Argument a2	Second input argument, subtracted from a1.
Argument a3	Output vector.

subview

Creates a new view object from a parent object with a selected subset of the data of the parent object accessed by the new view.

Vector Subviews.

```
vsip_vview_f* vsip_vsubview_f(
   const vsip_vview_f* a1,
   vsip_index a2,
   vsip_length a3);
```

```
vsip_cvview_f* vsip_cvsubview_f(
   const vsip_cvview_f* a1,
   vsip_index a2,
   vsip_length a3);
```

```
vsip_vview_i* vsip_vsubview_i(
   const vsip_vview_i* a1,
   vsip_index a2,
   vsip_length a3);
```

```
vsip_vview_vi* vsip_vsubview_vi(
   const vsip_vview_vi* a1,
   vsip_index a2,
   vsip_length a3);
```

```
vsip_vview_mi* vsip_vsubview_mi(
   const vsip_vview_mi* a1,
   vsip_index a2,
   vsip_length a3);
```

```
vsip_vview_bl* vsip_vsubview_bl(
   const vsip_vview_bl* a1,
   vsip_index a2,
   vsip_length a3);
```

Returns	Pointer to the new vector view object
Argument a1	Input vector.
Argument a2	Index of element in a1 starting the new vector view. The first element is index 0 (zero).
Argument a3	Length of new output vector view.

Matrix subviews.

```
vsip_mview_f* vsip_msubview_f(
   const vsip_mview_f* a1,
   vsip_index a2,
   vsip_index a3,
   vsip_length a4,
   vsip_length a5);
```

```
vsip_cmview_f* vsip_cmsubview_f(
   const vsip_cmview_f* a1,
   vsip_index a2,
   vsip_index a3,
   vsip_length a4,
   vsip_length a5);
```

Returns	Pointer to new matrix view object.
Argument a1	Input parent view.
Argument a2	Row index of parent view for first element in child view.
Argument a3	Column index of parent view for first element in child view.
Argument a4	Length of column (number of rows) of child view.
Argument a5	Length of row (number of columns) of child view.

sumsqval

Sum all the squares of the elements of a vector and return the sum.

```
vsip_scalar_f* vsip_sumsqval_f(
   const vsip_vview_f* a1);
```

Returns	Input vector elements squared and summed.
Argument a1	Input vector

sumval

Sum all the elements of a vector and return the sum. For boolean the number of true values is returned.

```
vsip_scalar_f* vsip_vsumval_f(
   const vsip_vview_f* a1);
```

```
vsip_scalar_vi* vsip_vsumval_bl(
   const vsip_vview_bl* a1);
```

Returns Sum of input vector values. For boolean the number of true values is returned.

Argument a1 Input Vector.

swap

Exchange the elements of two vectors.

```
void vsip_vswap_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

```
void vsip_cvswap_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1 Input and output vector.

Argument a2 Input and output vector.

tan

Tangent function.

```
vsip_vview_f* vsip_vtan_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2);
```

Argument a1 Input view.

Argument a2 Output view of Tangent values.

Ternary Functions

These functions involve two operations, and three inputs. The operations are a combination of add, multiply and subtract.

Vector vector add and vector multiply. Add two vectors elementwise and then multiply the result elementwise times a third vector.

```
void vsip_vam_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
```

```
void vsip_cvam_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector vector multiply and vector add. Multiply two vectors elementwise and then add the result elementwise to a third vector.

```
void vsip_vma_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
void vsip_cvma_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector vector multiply and scalar add. Multiply two vectors elementwise and then add the result elementwise to a scalar.

```
void vsip_vmsa_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_scalar_f* a3,
   const vsip_vview_f* a4);
void vsip_cvmsa_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cscalar_f* a3,
   const vsip_cvview_f* a4);
```

Vector vector multiply and vector subtract. Multiply two vectors elementwise and then subtract elementwise from the result a third vector.

```
void vsip_vmsb_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
void vsip_cvmsb_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector scalar add and vector multiply. Add a scalar to a vector elementwise and then multiply the result elementwise times a third vector.

```
void vsip_vsam_f(
   const vsip_vview_f* a1,
   const vsip_scalar_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
void vsip_cvsam_f(
   const vsip_cvview_f* a1,
   const vsip_cscalar_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector vector subtract and vector multiply. Subtract the second input vector from the first input vector multiply the result times a third input vector elementwise.

```
 void vsip_vsbm_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
```

```
void vsip_cvsbm_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector scalar multiply and vector add. Multiply a scalar times a vector and subtract a third vector from the result.

```
void vsip_vsma_f(
   const vsip_vview_f* a1,
   const vsip_scalar_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
```

```
void vsip_cvsma_f(
   const vsip_cvview_f* a1,
   const vsip_cscalar_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Vector scalar multiply and scalar add. Multiply a scalar times a vector and then add a scalar to the result.

```
void vsip_vsmsa_f(
   const vsip_vview_f* a1,
   const vsip_scalar_f* a2,
   const vsip_scalar_f* a3,
   const vsip_vview_f* a4);
```

```
void vsip_cvsmsa_f(
   const vsip_cvview_f* a1,
   const vsip_cscalar_f* a2,
   const vsip_cscalar_f* a3,
   const vsip_cvview_f* a4);
```

Argument a1	First input vector view.
Argument a2	Second input vector view or scalar.
Argument a3	Third input vector view or scalar.
Argument a4	Output vector view.

toepsol

Solve a Toeplitz linear system. The Toeplitz matrix must be symmetric if real or Hermitian if complex and positive definite. The matrix is square, and we solve a system of the form $\bar{T}\vec{x} = \vec{y}$. Since the Toeplitz matrix is completely determined by its first row, then only a vector view containing the elements of the first row is required for matrix input.

```
int vsip_toepsol_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3,
   const vsip_vview_f* a4);
int vsip_ctoepsol_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3,
   const vsip_cvview_f* a4);
```

Returns	Zero if successful, -1 if memory allocation failure, positive if the Toeplitz matrix is not positive definite.
Argument a1	Input vector containing first row of Toeplitz matrix. This vector may not overlap w any other vector in the work space.
Argument a2	Input vector containing $\vec{y}$. This vector may not overlap any other vector in the work space. This vector may be overwritten during the calculation.
Argument a3	Vector, the same length as a2, for scratch space. The data elements of the vector on input and output are not defined, and may be any value. This vector may not overlap any other vector in the work space.
Argument a4	Output solution vector $\vec{x}$. This vector may not overlap any other vector in the work space.

transview

Create a view of the transpose of a matrix.

```
vsip_mview_f* vsip_mtransview_f(
   vsip_mview_f* a1);
vsip_cmview_f* vsip_cmtransview_f(
   vsip_cmview_f* a1);
```

Returns	Pointer to created view, or NULL if the view creation failed.
Argument a1	Input matrix view.

xor

Performs a bitwise “exclusive OR” (“XOR”) operation between two integer views, or a logical “XOR” between two boolean views.

```
void vsip_vxor_i(
   const vsip_vview_i* a1,
   const vsip_vview_i* a2,
   const vsip_vview_i* a3);
```

```
void vsip_vxor_bl(
   const vsip_vview_bl* a1,
   const vsip_vview_bl* a2,
   const vsip_vview_bl* a3);
```

`Argument a1` Input view.

`Argument a2` Input view.

`Argument a3` Output view.

CHAPTER 3 Introduction to VSIPL Programming using the Core Lite Profile

Introduction

This chapter introduces programing methods using VSIPL in general, and the VSIPL Core Lite function set in particular. Although this chapter is written for the VSIPL Core Lite profile it works just as well as an introduction for the Core profile. The examples are simple. Explanatory text accompanies each example.

Support Functions

The support functions are those functions used to make or destroy VSIPL objects, copy data, modify object properties (such as stride, length and offset) or do input and output from VSIPL. The input and output functionality of VSIPL will be handled in its own section since it is a complicated topic.

Block Creation

The base method for block creation is the function **`blockcreate`**. This function takes a size argument, which indicates how many elements of the block type to create, and a VSIPL hint indicating how the data will be used by the program. TASP VSIPL Core Lite does not support this hint.

Generally examples will have a zero for creation hints since they are not supported by the TASP version of the library; however if the programer expects to develop code using TASP VSIPL on a workstation and then compile the workstation code on an embedded product, then it is recommended the programer use a VSIPL hint if it is supported by the embedded product. All VSIPL implementations are required to ignore the hint if it is not supported, so using an unsupported hint is harmless.

Vector Creation

The base method for vector creation is the **`bind`** function. This function creates a vector object, binds a block to the vector, and sets the stride, offset, and length of the vector to view the required data within the block. Example 2 below is a code segment to create a real float block and a complex float block and attach a vector to each block.

Example 2

```
1   /* Create a block and bind a vector to it */
2   vsip_block_f *a = vsip_blockcreate_f(10,0);
3   vsip_cblock_f *b = vsip_cblockcreate_f(10,0);
4   vsip_vview_f *v_a = vsip_vbind_f(a,0,1,10);
5   vsip_cvview_f *cv_b = vsip_cvbind_f(b,0,1,10);
```

Notice that we create a block in **lines 2** and **3** each of size 10 elements, but the elements of block **`a`** are real and the elements of block **`b`** are complex. In **lines 4** and **5** we define a real view **`v_a`** and a complex view **`cv_b`**. The view is created with the **`bind`** function, and we set the offset to zero, the stride to one, and the length to ten. Notice that the vectors we create here encompass the entire block, and could just as well have been created with the convenience **`create`** function used in Example 1.

Examination of the type definitions used for the block and vector views, and the function name of the **`blockcreate`** functions is worthwhile in order to develop an understanding of the VSIPL naming convention. The *precision* of the data here is float and is indicated with an **`_f`** prefix. The depth of the data is either real (understood in the name) or complex, indicated with a **`c`** prefix on the root name.

Other methods of view creation and view modification.

There are several methods of view creation. We will cover some of these in this section, and also some methods for view modification. It is frequently preferable to modify a view since no create needs to take place.

A new view of a block may always be created using the bind function. Each time this is done memory allocation takes place and the new view must be destroyed when no longer needed. Example 3 is how to use a current view and vector bind to create a new view. Lets say we want a vector view of every other element (element 0, 2, 4, etc.) of an available view.

Example 3

```
1   {
2      vsip_vattr_f attr;
3      vsip_vview_f *b;
4      vsip_vgetattrib_f(a,&attr);
5      b = vsip_vbind_f(attr.block,
6          attr.offset,2 * attr.stride,attr.length/2));
7      /* do something with b */
8      vsip_vdestroy_f(b);
9   }
```

We note that the vector view **`a`** resides outside the curly brackets. Since we don't know the stride and length of **`a`** we use the **`getattrib`** function in **line 4** to retrieve that information. In addition to offset, stride, and length getting the attribute structure also gets the block object. The **`bind`** function creates a view and binds the block to it. We set the offset to the same offset as **`a`**; however we only want every other point of the vector **`a`** so we set the stride to double the stride of **`a`**. Note that if we just set the stride to two we would get every other point in the block **`a`** was attached to, and not every other point of **`a`** (unless **`a`** happens to have a stride of one). Finally we set the length of the vector. We do not know if the length of **`a`** is even or odd, but length is some sort of unsigned integer, so division by 2 will result in an unsigned integer of the floor of the division, which is the number we need.

There are several ways to accomplish the same thing we accomplished in example 3. For instance in example 4 we demonstrate the same affect, but use **`cloneview`** instead of **`bind`** and for a change of pace we make the vector **`a`** complex.

Example 4

```
1   {
2      vsip_cvattr_f attr;
3      vsip_cvview_f *b = vsip_cvcloneview_f(a);
4      vsip_cvgetattrib_f(a,&attr);
5      attr.stride *= 2;
6      attr.length /=2;
7      vsip_cvputattrib_f(b,&attr);
8      /* do something with b */
10     vsip_cvdestroy_f(b);
11  }
```

Now we notice two things here. The first is VSIPL strides and offsets are in terms of the block element type. There is no difference in calculating the length and stride for this new complex view than there was in the real view of example 3. We also see a new function **`putattrib`** in **line 7**. Note that putting an attribute is the opposite of getting an attribute, except that the block value of the attribute is ignored. The block of the view object is set on view creation in **line 3**. The block attribute of a view is always set when the view is created, and it is not possible to reset the views block attribute. Also notice that the attribute is passed by reference (a pointer), both for getting, and putting, the attribute.

Another method to create a view is the **`subview`** function. The subview function takes an index into the parent view of the first element of the child view, and a length. Note that indexes are into vectors, not blocks. The stride is inherited from the original view, and there is no argument to allow resetting the stride in the **`subview`** function. In example 5 below we do example 4 again using a subview. This time we assume we know the stride and length of the vector **`a`**, and have stored them in variable **`a_stride`** and **`a_length`** respectively.

Example 5

```
{
   vsip_cvview_f *b = vsip_vsubview_f(a,0,a_length/2);
   vsip_vputstride_f(b,a_stride * 2);
      /* do something with b */
   vsip_cvdestroy_f(b);
}
```

Example 5 is shorter than example 4, but that is mostly because we already know the stride and length of the input vector. The new function to note here, besides **`subview`**, is the **`put-stride`** function on **line 3**. The Core Lite profile does not support any of the get attribute functions except **`getattrib`**, but it does support all the put attribute functions, including **`putlength`** and **`putoffset`**.

The final methods we will discuss to make views are **`realview`** and **`imagview`**. These two functions are so important that we will discuss them in their own section below.

Viewing the Real and Imaginary portions of a Complex Vector

In the elementwise function set there are two functions **`vsip_vreal_f`** and **`vsip_vimag_f`** which will copy the real or imaginary portion of a complex vector to a real vector, and another function **`vsip_vcmplx_f`** which will copy two real vector, one designated as real and one as imaginary, to a complex vector. Frequently it is desirable to operate on a complex vectors real or imaginary portions separately, but using the above function set is a lot of copying and requires extra memory allocation to allow room for the copies. What is really desirable is to be able to produce a real and imaginary view of the two parts of a complex vector in-place with no copies.

Functions in VSIPL allow one to create real and imaginary views of a complex vector. The functions are **`vsip_vimagview_f`** and **`vsip_vrealview_f`**. Producing a view of the real or imaginary part of a complex view is more involved than one might at first think. We will only look at one of the issues here. The problem is that these function create a vector view of type **`vsip_vview_f`**, a real vector view. This type view must be attached to a block of type **`vsip_block_f`**; however the complex view that the real views for the imaginary part and real part are *derived* from is bound to a complex block of type **`vsip_cblock_f`**. The first thing that must be done (by the implementation) is that a real block must be derived from the complex block which represents the data of the real or imaginary portion of the complex block. This block is termed a *derived block*.

A derived block is of the same data type as any other real block. Whether or not a block is derived from a complex block is a part of the state information kept by the block object. Derived blocks may not be destroyed. The derived block is destroyed when the complex block it is derived from is destroyed. The derived views are destroyed in the normal manner using **`vsip_vdestroy_f`**.

The only way to get a derived block is to derive a view (a *derived view*) using the **`imagview`** or **`realview`** functions. The method VSIPL uses to create the derived block is implementation dependent. These functions create a real view, bind the real view to the derived block, and set the offset, strides and lengths of the real view to view the required real or imaginary portion of the parent complex view. Although the length of the new view will be the same as the parent view, the stride and offset are implementation dependent. If these are needed for some reason the derived view must be queried.

Derived blocks may not be destroyed directly, they are destroyed when the parent complex block is destroyed. Derived blocks may be bound to new vector views. It is recommended that new views bound to derived blocks stay within the data space spanned by the original derived view.

There are some other subtle issues which we can ignore most of the time, and will ignore for this introduction. Lets do a couple of simple examples.

We may want to initialize a complex vector to zero. In the Core Lite profile there is no complex fill operation, only a real fill. Example 6 shows a method to fill a complex vector with a zero. Assume we have already produced the complex vector **a** outside the brackets.

Example 6

```
1   {  /* replacement for vsip_cvfill_f */
2      vsip_vview_f *b = vsip_vrealview_f(a);
3      vsip_vfill_f(b,0.0);
4      vsip_vdestroy_f(b);
5      b = vsip_vimagview_f(a);
6      vsip_vfill_f(b,0.0);
7      vsip_vdestroy_f(b);
8   }
```

It is important to notice that we destroy vector `b` twice, once on **line 4** and again on **line 7.** This is required. When we define the vector `b` on **line 2** we actually define a pointer of type real vector view. When we destroy the vector view in **line 4** we don't destroy the pointer, just what the pointer was pointed to which is the vector view created in **line 2**. We then create a new vector view in **line 5** and store the pointer in `b`. If we had not destroyed the object pointed to by `b` in **line 4** then in **line 5** we would have replaced the view object pointer and leaked the memory allocated for the real view object in **line 2**.

If this is clear, great. If not think about it this way. One wouldn't want to do

```
/* bad code */
float *b;
b = (float)malloc(N * sizeof(float));/* allocate memory */
b = (float)malloc(N * sizeof(float));/* leak above memory */
free((void *) b);
```

This is equivalent to what happens if you don't destroy a vector view before assigning a new vector view. Of course this is also true for blocks. With (VSIPL) blocks not destroyed properly you also leak the memory associated with the data array.

Another function that is not included with the Core Lite profile is the Euler function. Euler takes an input vector of angles (in radians) and outputs a complex vector of cosine values in the real part and sine values in the imaginary part. Example 7 shows us how to do that, and will also demonstrate some of the in-place functionality of VSIPL. In-place means to replace the input with the output. Most elementwise functions support in-place, but not all functions do. See the VSIPL specification for more details of in-place.

Example 7

```
1    vsip_cvview_f *v = vsip_cvcreate_f(N,0);
2    {/* do euler */
3       vsip_vview_f *v_r = vsip_vrealview_f(v);
4       vsip_vview_f *v_i = vsip_vimagview_f(v);
5       vsip_vramp_f(0,ft_f_2PI/(vsip_scalar_f)N,v_r);
6       vsip_vsin_f(v_r,v_i);
7       vsip_vcos_f(v_r,v_r);
8       vsip_vdestroy_f(v_r);
9       vsip_vdestroy_f(v_i)
10   }
```

In **line 1** of example 7 we create a complex vector of length `N`. We want to perform an Euler operation to fill this vector with sine and cosine values of angle arguments equally distributed between zero and 2π. In this example the input vector will stop just one increment short of 2π. In **lines 3** and **4** we produce views of the real and imaginary portion of the complex vector. In **line 5** we fill the real part of the vector `v` with angles starting at zero and incrementing by $(2\pi)/N$ until the last value of the vector `v_r` is $((N-1)/N)2\pi$. In **line 6** we place the sine of the real part values into the imaginary part of vector `v`. In **line 7** we replace the angles in the real part of the vector `v` with their cosine values. **Line 7** is the in-place operation. In **line 8** and **9** we destroy the views created in **lines 3** and **4** since they are no longer needed. Note that destroying these views does not destroy the data. The data is stored in the complex block and is still viewed by the complex vector `v.`

VSIPL Input and Output Methods

Since VSIPL blocks are created using incomplete type definitions it is not possible to manipulate the data array directly. There is no method within VSIPL to retrieve a pointer to any data memory which was created using `create` or `blockcreate.`Blocks created by these functions are termed VSIPL blocks.

It is important to get data into or out of VSIPL in order to communicate with other processes or to manipulate the data directly for some purpose. VSIPL has an understanding of owning the data it operates on. Now VSIPL blocks are always owned by VSIPL and are said to be in the *admitted* state. It is not possible to remove VSIPL blocks from the admitted state. It is possible to create a block and bind it to memory allocated by the application external to VSIPL. This type memory, and block, are termed *user data* arrays, and *user* blocks. A user block is created using the `blockbind` function. It is created in a *released* state. It is an error to use any VSIPL function which will read or write the data array of a released block.

When a user block is to be used by VSIPL It must first be admitted to VSIPL using the `blockadmit` function. When the application needs to access the data of an admitted user block directly the block must first be released using the `blockrelease` function.

Note the following. A user block and a VSIPL block of the same type and precision have the same type definition. All of the information as to whether a block is a *user* block, a *VSIPL* block, or is *released* or *admitted* is maintained by the block object as state information. A VSIPL block is always admitted, and may not be released. A user block is created as released and may be admitted or released as required.

Rebinding user data to a user block

In a situation where input and output (I/O) is continuous, sometimes called data streaming, it would be bad resource management to destroy and allocate new user blocks continuously where it is desirable to have multiple data buffers used for essentially the same thing. In addition the view setup on a user block would also need to be destroyed and reconstructed every time a new buffer is admitted to VSIPL and an old buffer is released. To get around this problem a function was defined in VSIPL that allows one to rebind a new buffer to a block. The new buffer has the same features as the old buffer. This way, for instance, while the second buffer is being filled VSIPL can be operating on data from the first buffer. When VSIPL is done the user block is released, and when the second buffer is filled it is bound to the user block using **`rebind`**, and the first buffer unbound from the block is free to accept new data from the user process while the user block is admitted to VSIPL for data manipulation.

I/O Example

In example 8 we assume we need an elementwise vector cosh function. In the VSIPL standard there is a general function for doing elementwise operations like this; however it is not included in the Core profile. For this example we would like to manipulate the elements directly and then import them into VSIPL. We will do this in-place.

Now we examine Example 8 below. Note the use of the capital **`IP`** in the vector cosh function name to denote in-place is the author's notation, not VSIPL's.

Example 8

```
#include<vsip.h>
int VU_vcoshIP_f(vsip_vview_f *a)
{  vsip_vattr_f attr;
   vsip_vview_f *b = a;
   vsip_scalar_f *buff;
   vsip_length n;
   vsip_block_f *B;
   vsip_vgetattrib_f(a, &attr);
   B = attr.block; n = attr.length;
   if ((buff = vsip_blockrelease_f(B,1)) == NULL){
      buff=(vsip_scalar_f *)malloc(
           n * sizeof(vsip_scalar_f));
      if(buff != NULL){
           if((B = vsip_blockbind_f(buff,n,0)) == NULL){
              free((void*)buff);
              return 0;
           }
         }else{ return 0;
      }
      vsip_blockadmit_f(B,0);
      b = vsip_vbind_f(B,0,1,n);
      vsip_vcopy_f_f(a,b);
      vsip_blockrelease_f(B,1);
      }
   while(n-- >0){
      *buff = cosh(*buff); buff++;
   }buff = vsip_blockfind_f(B);vsip_blockadmit_f(B,1);
   if(a != b){
      vsip_vcopy_f_f(b,a);
      vsip_valldestroy_f(b);
      free((void *) buff);
   }return 1;
}
int main()
{     int VU_vcoshIP_f(vsip_vview_f *);
      void VU_vprint_f(vsip_vview_f*);
      vsip_vview_f *A = vsip_vcreate_f(8,0);
      vsip_vramp_f(0,.2,A);
      printf("A = \n");VU_vprint_f(A);
      VU_vcoshIP_f(A);
      printf("cosh(A) = \n");VU_vprint_f(A);
      vsip_valldestroy_f(A);
      return 1;
}
```

```
44  void VU_vprint_f(vsip_vview_f* a){
45        int i;
46        vsip_vattr_f attr;
47        vsip_vgetattrib_f(a,&attr);
48        for(i=0; i<attr.length; i++)
49           printf("%6.4f ",vsip_vget_f(a,i));
50        printf("\n");
51        return;
    }
```

In **line 10** we attempt to release the block attached to vector **`a`**. If the block releases we go to **lines 25-26** where the buffer returned from the blockrelease is used directly to calculate an in place cosh. We then readmit the block in **line 27** and return a 1 for successful in **line 32**. Note we set vector **`b`** equal to **`a`** in **line 4** so **line 28** is false and we fall through to the return statement.

If the **`blockrelease`** returns a null pointer at **line 10** then we know that the input vector **a** is not a user vector. In this case we create a buffer of the proper size to hold the number of **`vsip_scalar_f`** elements in vector **`a`**. We check to make sure we were successful at allocating memory for a block, and return zero if not successful. In **line 14** we create a block and bind the buffer to it. In **line 20** we admit the block to VSIPL. In **line 22** we copy the input vector **`a`** to the user vector **`b`**. Note that pointer **`b`** was set to **`a`** in **line 5**, but was reset in **line 21** to the newly created user vector **`b`**. We now release the block **`B`** and go into the loop at **line 23** to calculate the cosh vector.

Note that we had to admit the block **`B`** before copying the input vector to the user vector, and then we had to release the block **`B`** before using the buffer directly. We now reset the **buff** pointer to its original value, and then admit **`B`** at **line 27**. Note that when we reset the pointer with **blockfind** on **line 27** we must do it before the block is admitted. An admitted block will not return the public buffer. If the pointer **`a`** is not equal to the pointer **`b`** then it has been reset. We enter the **`if`** code (at **line 28**) which copies the result of the cosh to the vector **`a`**, where we want it, and then on **line 30**, destroy the vector **b**, the block **B**, and any memory allocated by VSIPL. Note that the buffer **`buff`** was not allocated by VSIPL, but by the application. The application is responsible for cleaning that up. On **line 31** we free this memory.

Note that the admission on **line 20** has a zero as the second argument. This equates to false in VSIPL and tells VSIPL that we are not interested in what the buffer contains. Note that the release in **lines 10** and **23** and the admit in **line 27** have 1 for this argument. This equates to a true in VSIPL and indicates that the buffer contains data that we are interested in maintaining during the change of state of the block.

To test our code we do a simple program starting on **line 34**. In Matlab our program (main) looks like.

```
a = [0:.2:1.4]
 a =
       0    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000   1.4000
 a = cosh(a)
```

```
a =
   1.0000    1.0201    1.0811    1.1855    1.3374    1.5431    1.8107    2.1509
```

The output of Example 8 is

```
A =
0.0000 0.2000 0.4000 0.6000 0.8000 1.0000 1.2000 1.4000
cosh(A) =
1.0000 1.0201 1.0811 1.1855 1.3374 1.5431 1.8107 2.1509
```

Here we elected to use a *VSIPL* vector for our input to our cosh function; however we could have used a *user* vector and avoided the creates and destroys in the function.

Complex User Data

Most user data will be an array of vsip scalars of the type of the block the user array will be bound to. There are a few exceptions in VSIPL and only one exception in VSIPL Core Lite. This exception is an array of complex.

For user data, complex is **not** an array of type vsip_cscalar_f. For a complex user data array of size N the memory allocated is of type vsip_scalar_f, and is either two arrays of equal size N or a single array of size $2 \times N$. The first case of two arrays is termed split data, and the second case of a single array is termed interleaved data. For interleaved data the elements are organized consecutively as real, imaginary, real, imaginary, etcetera. For split data one array is real, the other imaginary. Examination of **`vsip_cblockbind_f`** will show how the two cases are handled in the function call.

Note that because of the possibility of split user data, requiring two data pointers, the I/O support functions for complex **`release`**, **`admit`**, and **`rebind`**, are slightly more complicated than their real counterparts. Examination of the prototypes in chapter two should make the differences clear.

There are many intricacies to I/O in VSIPL that were not covered in this section; however the author does not want to get bogged down in an introductory section with a lot of details. There is a great deal of information available in Appendix A, VSIPL fundamentals which deals with user blocks and data in some depth.

Scalar Functions

The Core Lite profile only defines four scalar functions. Three of these are **`vsip_real_f`**, **`vsip_imag_f`**, and **`vsip_cmplx_f`**. There is also a function **`vsip_CMPLX_f`** which is included to allow the vendors to include a macro for creating complex numbers. The TASP VSIPL Core Lite profile has not implemented **`CMPLX`** as a macro, and just calls the **`cmplx`** function within **`CMPLX`**.

The functions **`real`**, **`imag`** and **`cmplx`** are important for manipulating complex scalars. For example **`vsip_cvget_f`** returns a scalar of type **`vsip_cscalar_f.`** To extract the real or imaginary portion of this scalar you would use **`vsip_real_f`** or **`vsip_imag_f`**. To make a complex number you would use **`vsip_cmplx_f`**. For instance:

```
/* put a complex number (a,b) at */
/* element number 6 (index #5) in a vector */
vsip_vput_f(complex_vector,5,vsip_cmplx_f(a,b));
```

Or to print the real and imaginary portions of a complex number:

```
vsip_cscalar_f a;
/* some code */
a = vsip_cvget_f(complex_vector,5);
print(“%f + %fi ”,vsip_real_f(a),vsip_imag_f(a));
```

VSIPL Elementwise Functions

Most elementwise functions are straightforward. Generally these functions take one or more input vector views, or a scalar and a vector view, and do an element by element calculation outputting the result in an output vector. A few of the elementwise functions, such as **`sumval`** and **`dot`** calculate some value based on an elementwise operation and accumulate. Finally we have **`ramp`**, **`fill`**, and random numbers. These functions generate data and fill a vector based on some formula, element by element. The functions **`ramp`** and **`fill`** have been used in previous examples and are easy to understand based on their prototype definition; however random number generation is more complicated and requires some explanation.

Except for random number generators no specific example of elementwise functions will be included; however almost all of the examples include elementwise operations.

Random Number Generation

The VSIPL random number generator is a more complicated function, actually a set of functions, than the other elementwise functions. The function set includes a create function which is used to create a random number state object, a destroy function to destroy the random state object when we are done with it, and a vector random generator function. For VSIPL Core Lite only the uniform generator for real vectors is part of the profile. The function generates a uniform random number between zero and 1.

The vector random number is simple, just filling a vector with uniform random numbers from the sequence, and updating the state object each time.

The random number state creation is a little more complicated than the generator function. To understand it first one must understand the expectation that VSIPL programs will be run in multiple processor environments, and it is desirable to be able to calculate independent random number sequences on the different processes based on a single seed value. To this end the **`randcreate`** function has an argument which indicates the total number of processes that will be calculating random sequences, and an argument that indicates which process the state object is being created for. Having the value of the total number of processes allows the create function to subset the random number sequence into the proper number, and having the number of the process allows the create to initialize the state object for the process to the correct subset.

The random state creation also allows one to chose either the required portable random number generator (portable because all implementations use the same generator, defined by

VSIPL), or a non-portable generator of the implementors choosing. There is no requirement for an implementation to support its own generator, and the non-portable generator may default to the portable version.

Below find Example 9 demonstrating the **`randu`** function set. In this example create two separate vectors full of independent random numbers. The random number generator is also used in Examples 10 and 11 to make some data to work with.

Example 9

```
1   /* Create two independent random sequences */
2   #include<stdio.h>
3   #include<vsip.h>
4   #define TYPE VSIP_NPRNG /* non portable generator flag*/
5   #define N 1024  /* length of random vector */
6   #define init 17 /* random initialization */
7
8   int main()
9   {
10        vsip_vview_f *ran1 = vsip_vcreate_f(N,0),
11                     *ran2 = vsip_vcreate_f(N,0);
12        vsip_randstate *state1 =
13                 vsip_randcreate(init,2,1,NPRNG);
14        vsip_randstate *state2 =
15                 vsip_randcreate(init,2,2,NPRNG);
16        vsip_vrandu_f(state1,ran1);
17        vsip_vrandu_f(state2,ran2);
18  /* do something with ran1 and ran2 */
19        vsip_randdestroy(state1);
20        vsip_randdestroy(state2);
21        vsip_valldestroy_f(ran1);
22        vsip_valldestroy_f(ran2);
23        return 1;
24  }
```

Example 9 is very simple and does not do anything interesting. In **lines 10** and **11** we create two vectors to hold our random sequences. In **lines 12** and **14** we create two random state objects. Argument two of rand create tells the randcreate function we want a state object suitable for *two* independent random number generators. Argument 3 of **line 12** says to create the state object for the *first* process. Argument 3 of **line 14** says to create the state object for the *second* process. In **lines 16** and **17** we generate the random numbers and fill the two vectors created to hold them. **Lines 19** through **22** clean up all the objects created by VSIPL.

Signal Processing Functions

The Core Lite profile supports a histogram function, a complex to complex out of place Fourier transform, a real to complex, and a complex to real Fourier transform, and a finite impulse response (FIR) filter with desampling.

The Fourier Transform

The fourier transform function set for Core Lite includes three discrete fourier transforms (DFT), three corresponding create functions to create the fourier transform object, and a destroy function to destroy the fourier transform object. There is only one data type for the fourier transform object, and so only one destroy function is required. The type of fourier transform object created (for **`ccfftop`**, **`rcfftop`** or **`crfftop`**) is kept as state information by the fourier transform object.

The **`ccfftop_create`** function will create an fft for either a forward or inverse DFT. It is assumed that **`rcfftop`** and **`crfftop`** are done in the forward and inverse directions respectively, so there is no direction arguments in **`rcfftop_create`** and **`crfftop_create`**. Within the header file **`vsip.h`** resides an enumerated type definition which may be used for the direction argument.

```
typedef enum{
   VSIP_FFT_FWD = -1,
   VSIP_FFT_INV = 1
}   vsip_fft_dir;
```

VSIPL requires an FFT (Fast Fourier Transform) algorithm for a radix of two with one radix of 3 if necessary. Any length DFT is required to be supported, but a fast transform is only required for lengths of $N = 2^n 3^m$ where m is either 0 or 1. The TASP VSIPL Core Lite FFT base algorithm actually supports $N = 2^{m_0} 4^{m_1} 8^{n_0} 3^{n_1} 5^{n_2} 7^{n_3}$ where m_i is either 0 or 1 and n_i is some integer greater than or equal to zero. In the TASP VSIPL implementation if N is not factorable as above then the last factor (which is not factorable by 2, 4, 8, 3, 5, or 7) is used as a final factor and a DFT is done for that stage.

The functions **`rcfftop`** and **`crfftop`** require that the data be even, and only half the transform is returned. So for **`rcfftop`** the input real vector is of length N and the output complex vector is of length $N/2 + 1$. The inverse is true for **`crfftop`**.

So far we have done a few meaningless examples to illustrate VSIPL. This example will be just as meaningless. Lets find the FFT of a real vector of random numbers using rcfftop, extend the FFT to full length and find its inverse using **`ccfftop`**. We will then subtract the input vector from the output vector and find the mean square value of the result, which should be close to zero.

Example 10

```
#include<stdio.h>
#include<vsip.h>
#define RNL 1024  /* length of random vector */
#define RNS 17 /* random number seed */
#define RNT VSIP_PRNG /* random number type */
int main(){
   vsip_cvview_f *fft = vsip_cvcreate_f(RNL, 0),
                 *invfft = vsip_cvcreate_f(RNL, 0);
   vsip_randstate *state = vsip_randcreate(RNS,1,1,RNT);
   vsip_vview_f   *input = vsip_vcreate_f(RNL, 0);
   vsip_fft_f *rcfft = vsip_rcfftop_create_f(RNL,1,0,0);
   vsip_fft_f *ccfftI = vsip_ccfftop_create_f(
            RNL, 1.0/RNL, VSIP_FFT_INV,0,0);
   vsip_vrandu_f(state,input);
   vsip_cvputlength_f(fft,RNL/2+1);
   vsip_rcfftop_f(rcfft, input, fft);
   vsip_cvputoffset_f(fft, 1);
   vsip_cvputlength_f(fft, RNL/2-1);
   { /* fill out the forward fft to full length */
      vsip_cvview_f *temp = vsip_cvcloneview_f(fft);
      vsip_cvattr_f at; vsip_cvgetattrib_f(temp,&at);
      at.offset = RNL - 1; at.stride = - 1;
      vsip_cvputattrib_f(temp,&at);
      vsip_cvconj_f(fft,temp); vsip_cvdestroy_f(temp);
   }
   vsip_cvputoffset_f(fft,0);
   vsip_cvputlength_f(fft,RNL);
   vsip_ccfftop_f(ccfftI,fft,invfft);
   { /* compare results */
      vsip_vview_f *real = vsip_vrealview_f(invfft);
      vsip_vview_f *result = vsip_vimagview_f(invfft);
      vsip_vsub_f(input,real,result);
      printf(“%f\n”,vsip_vsumsqval_f(result)/RNL);
      vsip_vdestroy_f(real); vsip_vdestroy_f(result);
   }
   vsip_fft_destroy_f(rcfft);
   vsip_fft_destroy_f(ccfftI);
   vsip_randdestroy(state);
   vsip_cvalldestroy_f(fft);
   vsip_cvalldestroy_f(invfft);
   vsip_valldestroy_f(input);
   return 1;
}
```

In Example 10 we create the data space with `create` functions. We modify the stride, length and offset attributes with full knowledge of the initial attributes of the views. For this reason we don't need to get the attributes first. We do need to keep track as we move through the code however.

In **line 4** we define a constant for initializing the random number state created in **line 9.** In **line 14** we use the random number generator to initialize a real input vector of random values. In **line 7** and **8** we create two complex vectors of length RNL, the first (`fft`) to hold the transform of the random input vector, and the second (`invfft`) to hold the inverse transform of `fft`.

We want to do the forward transform using `vsip_rcfftop_f`. We made `fft` of length RNL since we plan on filling out the vector to a full length `fft` for use in `vsip_ccfftop_f` for doing the inverse transform; however `rcfftop` requires a vector of length $\mathrm{RNL}/2+1$ so we set this length in **line 25**. We create the fft object in **line 11** for a length of RNL and a scale factor of 1. The last two arguments of the create are not used in TASP VSIPL so we set them to zero.

In **line 16** we do the Fourier transform on `input` placing the result in `fft`. Now we need to select the redundant portion of fft in preparation for filling out the fft vector to full length. In **line 17** we set the offset of `fft` to 1 (the second element), since the first element is the DC value of the transform and is unique. The last value of the transform is also unique since the input real vector was even, so we set the length of `fft` to $\mathrm{RNL}/2-1$. Now we enter a section of code between **lines 19** and **25** where we copy the conjugate of the redundant section in reverse order to the end of the final transform vector. In `line 20` we create a clone of `fft` which sets the length of the vector properly. This is the vector we are going to copy into, and the first element of `fft` (as the view is currently defined) must copy to the first element of `temp`. We want this to be the last element of the block, so we set the offset of `temp` to the end of the block at offset $\mathrm{RNL}-1$. We want `temp` to travel backward through the block so we set the stride of `temp` to -1. We now do the copy and conjugation in one step using `cvconj` in **line 24**, and then destroy the `temp` vector in **line 24**. In **line 26** and **27** we restore the `fft` view to the entire block. What we have done between **lines 17** and **27** would look in Matlab, for Matlab vector a of length RNL, as

>> a(end:-1:RNL/2+2) = conj(a(2:RNL/2));

In **line 12** we create an fft object for use in `ccfftop` to do an inverse Fourier transform of length RNL and with a scale factor of $1/\mathrm{RNL}$. Notice that `ccfftop_create` has an argument for forward or inverse transform, but `rcfftop_create` does not have the argument and always goes in the forward direction. The last two arguments are not implemented in the TASP implementation of core lite and so are set to zero. We now find the inverse of `fft` in **line 28**.

In **lines 29** through **35** we subtract the `input` from the `real` output and examine the mean square value of the `result`. This should be very close to zero as the input and output should be the same.

The Finite Impulse Response Filter

The FIR (finite impulse response) function set is designed to allow for continuous filtering with desampling. The filter object saves state information from the previous filter operation allowing for continuous filtering of vector segments of a data stream. When desampling the number of returned filtered elements may vary depending upon the state of the filter object. For this reason the FIR filter function returns an integer which is equal to the number of elements in the output vector from the filter operation. Since a filter object is created there is a filter destroy function available for destroying the filter object.

The FIR filter create requires an input of a filter kernel, filter symmetry information, the length of the input vector to be filtered, and a desampling factor. The final two arguments are included to allow the vendor to optimize his routine for various common filter operations. Neither arguments are implemented in TASP VSIPL and so a zero will normally be placed here when using TASP VSIPL. For application programmers who are developing on TASP VSIPL for other hardware it is recommended to use the proper values for that hardware. TASP VSIPL ignores these values and will work fine for any information inserted.

The *filter kernel* and the symmetry argument vary depending upon the type of filter coefficients. For TASP VSIPL including all of the filter coefficients with a VSIP_NONSYM symmetry argument will always work. If the filter coefficients are symmetric and there are an even number then using the first half of the filter coefficients as the kernel and the VSIP_SYM_EVEN_LEN_EVEN argument will work. If the filter coefficients are symmetric and there are an odd number then using the first half of the filter coefficients plus the middle point ($(N-1)/2+1$) as the kernel and the VSIP_SYM_EVEN_LEN_ODD argument will work. TASP VSIPL always expands the filter coefficients to full length and does the same filter for all three cases, so the author recommends always using the VSIP_NONSYM version kernel, unless developing for another platform. Note that the *kernel* is simply a vector of filter coefficients as described above. Also be aware that the current FIR filter in TASP VSIPL is not a fast fir, and is not optimized in any way. For some operations it may be desirable to do the FIR filter directly using other VSIPL operations.

The FIR filter function requires a FIR filter object, an input vector of length N, and an output vector which has a length equal to the input vector length divided by the desampling factor rounded up to the nearest integer. This is commonly called the ceiling of N/D where N is the input vector length and D is the desampling factor.

The FIR is demonstrated in Example 11. The filter coefficients are in **lines 21** through **29** and are input to a user data array. In **line 31** we create a block and bind the coefficients to it using **`blockbind`**, and then **`bind`** the block to a vector view at **line 32**. Since this is a *user* block in **line 34** we **`admit`** it to VSIPL so we can use any vector views binding the block in our functions.

We then create the FIR object in **line 35**. Notice that we included the entire set of filter coefficients and so we create the object as VSIP_NONSYM. The actual filter coefficients are odd symmetric so we could have created the filter object as just the first 22 (of 43) coefficients and passed a symmetry argument of VSIP_SYM_EVEN_LEN_ODD. We then destroy the kernel since we no longer need it after the filter object is created. Notice we use **`alldestroy`** so that both the vector view and the block is destroyed. The kernel data is not destroyed because the blocks *user* state is set, and the destroy function knows to not destroy the *user* data array.

For the example we set a decimation factor D, average **avg**, and base length N in **lines 3** through **5**. In **line 6** we set a constant to initialize the random number generator.

We create an input vector of length $D \times N$ in **line 12**. This ensures the output from the FIR filter will be of length N. In **line 41** we fill the input vector with uniform random numbers between 0 and 1. In **line 42** we do a negative DC offset of our input vector by 0.5 and in **line 43** we filter the input vector into the output vector. In **line 44** and **45** we do an FFT estimate of the spectrum. Note power normalization is not done and is not important to the example. In **line 46** we do a running sum of the spectrums. In **line 4** we define **avg**, the number of sums, and in **line 48** we normalize our sum by the **avg** number. We then print out the result using a print subroutine VU_vprint_f contained at the end of the example. Note that **line 49** allows the output to be brought into Matlab and plotted by piping standard out into a "*.m*" file and executing the file in Matlab. (The author knows there are better ways to get data into Matlab, but he is too lazy to figure them out.)

In Figure 1 we note the frequency response of the filter coefficients as the bottom plot of the figure. Matlab code to generate this plot, and the filter coefficients for the example, were obtained from an internet site at Rice University
(*http://jazz.rice.edu/software/RU-FILTER/cpm/*)
There is a paper describing the method for calculating the coefficients by I. W. Selesnick and C. S. Burrus,
"Exchange Algorithms that Complement the Parks-McClellan Algorithm for Linear-Phase FIR Filter Design".
I. W. Selesnick appears to be the author of the Matlab code.

Summary

In this chapter we have quickly covered the functionality of the Core Lite profile and given examples on its use. We cover methods for creating and destroying blocks and views, and for obtaining views of the real and imaginary portions of complex views. *User* data and methods to get data in and out of VSIPL are discussed. Finally we do examples illustrating the use of the random number generator, Fourier transform, and the FIR filter.

Example 11 (Page 1 of 2)

```
#include<stdio.h>
#include<vsip.h>
#define N   1024
#define avg 1000
#define D      2
#define RNS  17       /* Random Number Seed */
#define RNT VSIP_PRNG /* Random Number Type */
void VU_vprint_f(vsip_vview_f*);
int main()
{
   int i;
   vsip_vview_f *dataIn   = vsip_vcreate_f(D * N,0);
   vsip_cvview_f *dataFFT = vsip_cvcreate_f(N/2 + 1,0);
   vsip_vview_f *dataOut  = vsip_vcreate_f(N,0);
   vsip_vview_f *spect_avg = vsip_vcreate_f(N/2 + 1.0,0);
   vsip_vview_f *spect_new = vsip_vcreate_f(N/2 + 1.0,0);
   vsip_randstate *state  = vsip_randcreate(RNS,1,1,RNT);
   vsip_fir_f *fir;
   vsip_fft_f *fft = vsip_rcfftop_create_f(N,1,0,0);
   vsip_scalar_f b[] =
            {0.0234, -0.0094, -0.0180, -0.0129,  0.0037,
             0.0110, -0.0026, -0.0195, -0.0136,  0.0122,
             0.0232, -0.0007, -0.0314, -0.0223,  0.0250,
             0.0483, -0.0002, -0.0746, -0.0619,  0.0930,
             0.3023,  0.3999,  0.3023,  0.0930, -0.0619,
            -0.0746, -0.0002,  0.0483,  0.0250, -0.0223,
            -0.0314, -0.0007,  0.0232,  0.0122, -0.0136,
            -0.0195, -0.0026,  0.0110,  0.0037, -0.0129,
            -0.0180 ,-0.0094,  0.0234};
    {
        vsip_block_f *kblock = vsip_blockbind_f(b,43,0);
        vsip_vview_f *kernel =
            vsip_vbind_f(kblock,0,1,43);
        vsip_blockadmit_f(kblock,1);
        fir = vsip_fir_create_f(kernel, VSIP_NONSYM,
            D * N, D,VSIP_STATE_SAVE, 0, 0);
        vsip_valldestroy_f(kernel);
     }
```

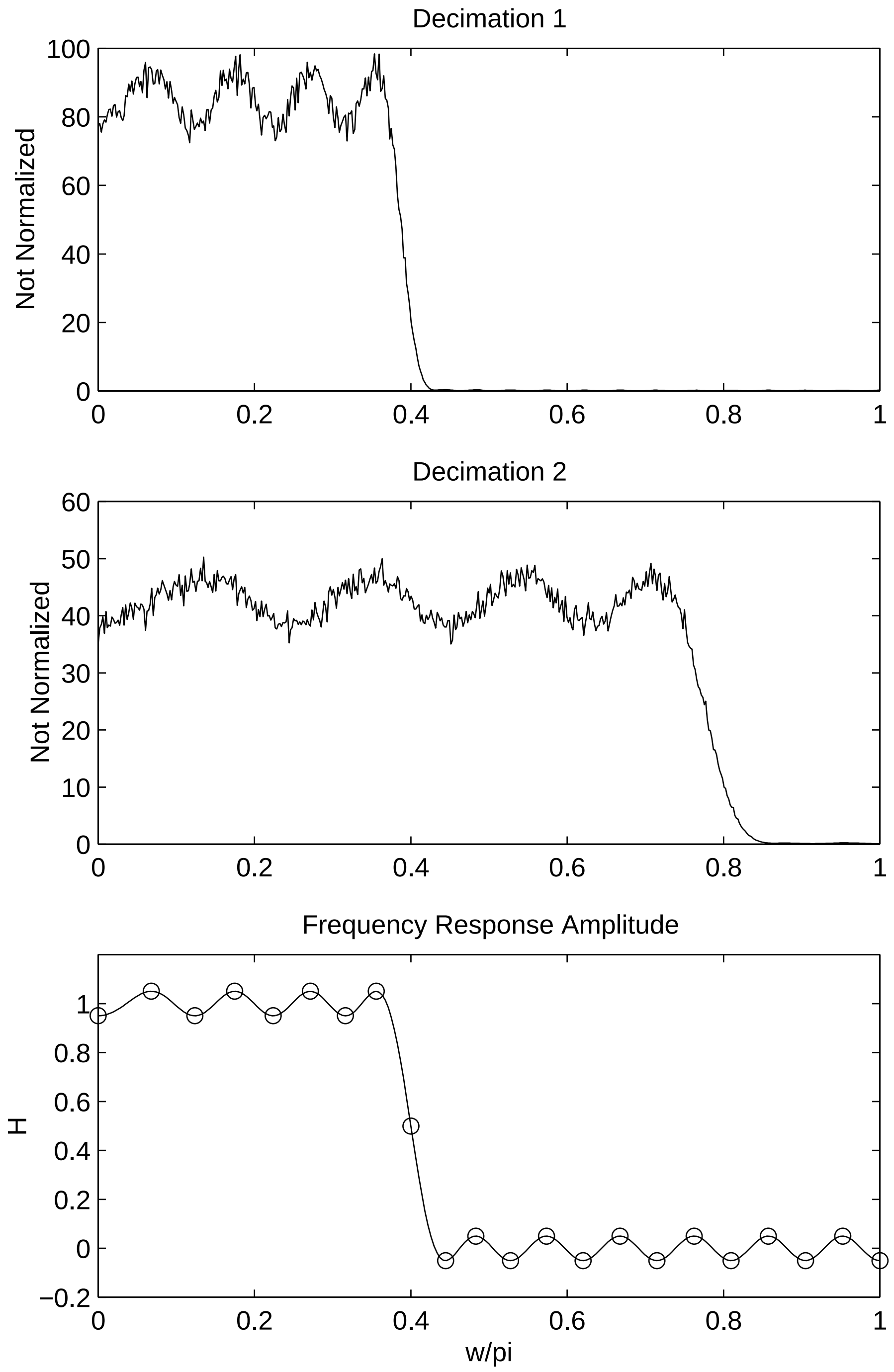
Example 11 (Page 2 of 2)

```
     vsip_vfill_f(0,spect_avg);
     for(i=0; i<avg; i++){
        vsip_vrandu_f(state,dataIn);
        vsip_svadd_f(-.5,dataIn,dataIn);
        vsip_firflt_f(fir,dataIn,dataOut);
        vsip_rcfftop_f(fft,dataOut,dataFFT);
        vsip_vcmagsq_f(dataFFT,spect_new);
        vsip_vadd_f(spect_new,spect_avg,spect_avg);
      }
      vsip_svmul_f(1.0/avg,spect_avg,spect_avg);
      printf(“spect_avg =”);VU_vprint_f(spect_avg);

      vsip_valldestroy_f(dataIn);
      vsip_valldestroy_f(spect_avg);
      vsip_valldestroy_f(spect_new);
      vsip_cvalldestroy_f(dataFFT);
      vsip_valldestroy_f(dataOut);
      vsip_randdestroy(state);
      vsip_fft_destroy_f(fft);
      vsip_fir_destroy_f(fir);
      return 1;
}

void VU_vprint_f(vsip_vview_f *a)
{
    vsip_vattr_f attr;
    vsip_index i;
    vsip_vgetattrib_f(a,&attr);
    printf(“[“);
    for(i=0; i<attr.length-1; i++)
         printf(“%7.4f;\n”,vsip_vget_f(a,i));
    printf(“%7.4f];\n”, vsip_vget_f(a,i));
    return;
}
```

Figure 1



The figures above are related to Example 11. The bottom plot is the frequency response related to the kernel in **lines 20** through **29** of the example. Matlab code available at internet site **jazz.rice.edu/software/RU-FILTER/cpm/** was used to generate the plot and the filter coefficients, and is directly from the first **Example** referenced on that web page. The plots Decimation 1 and 2 are the result of Example 11 for the stated decimation factors.

CHAPTER 4 Introduction to VSIPL Matrices

Introduction

In the previous chapter VSIPL programing was introduced using the Core Lite profile. One of the main differences between the Core Lite profile and the Core profile is the addition of matrix functionality in Core. This chapter will introduce VSIPL matrices.

Matrix Fundamentals

Although matrices are probably well understood by most readers the VSIPL forum had many, sometimes heated, discussions on how best to describe them within the VSIPL framework. Most of these discussions boiled down to terminology, and what is standard. In addition there were efforts to generalize matrices so the terminology would fit with higher order views so that the VSIPL terminology would be the same for three dimensional views as for two dimensional, or even one dimensional view.

Since the VSIPL Core Profile has no dimensions above two the author will stick with standard linear algebra matrix terminology. First we define some of the “standard terminology” as the author has found that not all people agree on what it is. Readers may not agree that this is standard terminology, but at least they will know what the author thinks.

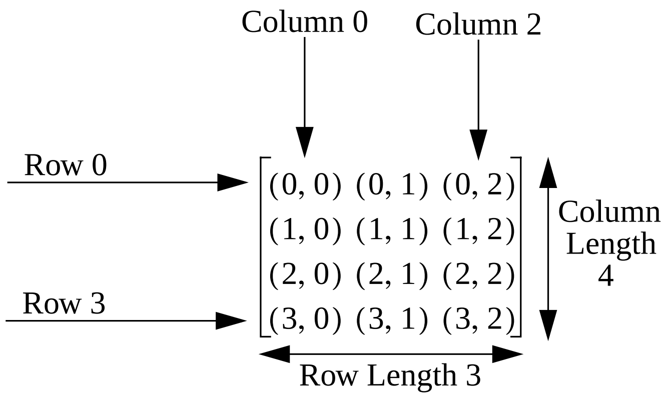
Some of the following may seem a bit fundamental. However it is important that we understand how matrices are described on a block, and how to manipulate the matrix view attributes properly. Without agreeing on the fundamentals it is easy to become confused, and the author remembers being very confused a few times.

A Matrix

A matrix is a set of data elements described by two indices. The first index is called a row index, and the second index is called a column index. When the elements of the matrix are placed on a piece of paper so that all the elements with the same row index go across the paper (in a row), and all the elements with the same column index go down the paper (in a column) and the rows with the smaller index values are above the rows with the higher index value (in order) and similar for the columns, then we have written the matrix down. VSIPL defines index values as starting at zero so that the matrix index of the very first element located at the top left of the matrix will be at $(0, 0)$.

The size of a matrix is usually described as m by n where m is the number of rows and n is the number of columns. Here is where confusion can begin. The number of rows in a matrix is also the number of elements in a column. The number of columns in a matrix is also the number of elements in a row. So a matrix size of $[m \text{ rows}, n \text{columns}]$, is also a matrix size of $[\text{column length } m \text{ , row length } n]$.

In general the author will generally speak of a matrix as having a certain column length and row length. The row length (number of elements in a row), and the column length (number of elements in a column) are two of the attributes of a matrix *view* used to define the matrix layout on a *block*. The following example of a matrix of size $(4 \text{ by } 3)$ may help.



Matrix Views

The underlying *block* containing data associated with a *matrix view* is the same type block as that associated with a *vector view.* The view on the block is what makes the object a vector or a matrix.

For a vector view the data is described using an offset from the beginning of the block to the first element of the vector, a stride through the block between elements in the vector, and a vector length which is the number of elements in the vector.

A *matrix view* is similar to a *vector view* except there are now two strides, and two lengths. The two strides are a row stride specifying the distance through the block between consecutive elements of a row, and a column stride specifying the distance through the block between consecutive elements in a column. The two lengths are a row length specifying the number of elements in a row of the matrix, and a column length specifying the number of elements in a column of a matrix. There is also an offset specifying the number of elements from the beginning of the block to the first element in the matrix.

The following may help in understanding how a matrix view maps the elements of a block into a matrix. For this example lets map a block of length 12 into a matrix of size 3,4. This is a dense mapping so that every element of the block is mapped into the matrix. We will do the first mapping so that the row stride is one, and the second mapping so that the column stride is one. The dimension with the smallest stride is called the *major* direction. We have the matrix

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}.$$

For this matrix we know the row length is 4 and the column length is 3. Note that the row length and the column length are properties of the matrix definition. These two attributes define the **size** and **shape** of the matrix. They come from a linear algebra point of view, and are important when doing matrix operations. Except for defining the minimum block size that a matrix will fit into and the minimum stride between elements in the non-major (minor) direction matrix lengths are not important when mapping a matrix on a block.

For the case of a row stride of one (row major) we have the following consecutive elements in the block.

$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{2,0}, a_{2,1}, a_{2,2}, a_{2,3}$

The offset and stride attributes define the location of the matrix in the block. They have nothing to do with a matrix from a algebra point of view, and are not needed when thinking about how matrices combine when doing matrix math. We note that since the matrix is dense the column stride is the row length. In the case above this is four. We can get this by counting the number of elements between $a_{o,o}$ and $a_{1,0}$.

Now lets consider the case of column major. For this case the column stride will be one and (in consecutive block locations starting at zero) the matrix layout will be

$a_{0,0}, a_{1,0}, a_{2,0}, a_{0,1}, a_{1,1}, a_{2,1}, a_{0,2}, a_{1,2}, a_{2,2}, a_{0,3}, a_{1,3}, a_{2,3}$.

Once again the matrix is dense so the row stride is the column length which is three.

We do one more case with this example for a block size of 18. We designate elements in the block which are not mapped by the matrix with an e_i. The matrix is row major, with a row stride of one, a column stride of five, and an offset of two. The matrix is not dense. The mapping looks as follows.

$e_0, e_1, a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, e_6, a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, e_{11}, a_{2,0}, a_{2,1}, a_{2,2}, a_{2,3}, e_{16}, e_{17}$

Matrix Creation

There are two fundamental methods for creating a matrix. The most fundamental method is to obtain a block object, either by creating it or by using one already created, of sufficient size to hold the matrix. Then a matrix is defined on the block using the matrix bind function (**`vsip_mbind_f`**). This looks like the following.

```
/* create a block and bind a matrix to it of size
   M rows by N columns */
vsip_block_f *block = vsip_blockcreate_f(M * N, VSIP_MEM_NONE);
vsip_mview_f *matrix =
       vsip_mbind_f(block,offset,
         col_stride,col_length,
         row_stride,row_length);
```

In the code segment above we have created a block exactly the size needed for a matrix of size M, N. This means that (in order for the matrix to fit in the block and make sense) the offset must be zero and either the column stride must be one and the row stride must be the column length, or the row stride must be one and the column stride must be the row length. A better way to create a dense matrix of this type is to use the matrix create function. This looks like the following.

```
/* Create a dense row major matrix of size M rows by
   N columns */
vsip_mview_f *matrix = vsip_mcreate_f(M,N,
                         VSIP_ROW,VSIP_MEM_NONE);
```

It is also possible to create a new matrix view from an existing matrix view using the matrix subview function. For complex matrices a real matrix view of the real or imaginary part of the complex view may be created using the matrix real view and imaginary view functions.

Extracting Vector views from Matrix Views

The ability to obtain a vector view of a row, column, or diagonal of a matrix is an important feature of VSIPL. The view can be created directly by using the bind function to create and bind a vector view to the same block the matrix is bound to, and then setting the attributes of the vector view to map the proper portion of the matrix view into the vector view, or it can be done with vector view create functions designed for this purpose. For example to obtain a vector view of the second row of a $5, 4$ matrix would look as follows.

```
/* Create a dense column major matrix of size 5,4
   and then create a vector view of the second row
   of the matrix. */
vsip_mview_f *matrix = vsip_mcreate_f(5,4,
                         VSIP_COL,VSIP_MEM_NONE);
vsip_vview_f *vview = vsip_mrowview_f(matrix,1);
```

To obtain a view of the first diagonal above the main diagonal of the matrix would look as follows.

```
/* Create a dense column major matrix of size 5,4
   and then create a vector view of the first upper diagonal
   of the matrix. */
vsip_mview_f *matrix = vsip_mcreate_f(5,4,
                               VSIP_COL,VSIP_MEM_NONE)
vsip_vview_f *vview = vsip_mdiagview_f(matrix,1);
```

Note that the index for the diagonal view is positive for diagonals above the main diagonal, and negative for diagonals below the main diagonal. The index data type for VSIPL is some type of unsigned integer and all indices start at zero and increase so this is a small problem. For this reason the data type for the diagonal index argument of the diagonal view function is a stride type, not an index type.

Fundamental Matrix Calculation

The author defines the fundamental matrix calculation as the calculation necessary to find the block location of any element in a matrix given the matrix index of the element and the stride and offset attributes of the matrix view. Given a matrix index (r, c) where r is the row and c

is the column, and matrix attributes o, r_s, c_s where o is the offset in the block to matrix element $(0, 0)$, r_s is the row stride, and c_s is the column stride we calculate the element offset in the block, designated $E_o(r, c)$, using the following formula.

$$E_o(r, c) = o + (r)(c_s) + (c)(r_s)$$

For the case of a dense matrix with stride in the major direction of one, then the stride in the minor direction will be equal to the length along the major direction. For other cases the necessary stride in the minor direction is not obvious. The following formula will give the minimum stride length in the minor direction to give a usable matrix. Let r_l be the row length and c_l be the column length. For row major we calculate the minimum c_s using

$$c_{s\min} = (r_l - 1)r_s + 1$$

and for column major we calculate the minimum r_s using

$$r_{s\min} = (c_l - 1)c_s + 1 .$$

Note that $(r_l - 1)$ is the largest possible index in a row, and $(c_l - 1)$ is the largest possible index in a column.

Simple Matrix Manipulations

The VSIPL core profile has many support functions for creating various sub views of a matrix, either a matrix subview or a vector view. Although these functions are easy to use, they may not be the most efficient means to an end. In addition VSIPL core does not support matrix elementwise functions. One could always do elementwise calculations using the element get (**`vsip_mget_f`**) and the element put (**`vsip_mput_f`**). This is very inefficient and is not a good way to do elementwise calculations. It is better to find vector views that view all or part of a matrix views data and then do the elementwise calculations using vector elementwise functions. In this section we explore some of the possibilities. We will use **`vsip_vadd_f`** as an example of an elementwise operation, but any elementwise function could be substituted.

A Simple Print Function

In order to make the examples usable we need to print the results. Below is a simple print utility which will allow printing (small) matrices in a format suitable for pasting into Matlab. We use this function (or a similar function) in examples below. Note in **lines 3** and **4** the use of the support functions to get the row or column length of a matrix, and in **line 10** the method to get a matrix element.

```
#include<vsip.h>
void VU_mprintm_f(char format[],vsip_mview_f *X)
{ vsip_length RL = vsip_mgetrowlength_f(X),
              CL = vsip_mgetcollength_f(X),
              row,col;
  vsip_scalar_f x;
  printf("[\n");
  for(row=0; row<CL; row++){
    for(col=0; col<RL; col++){
      x=vsip_mget_f(X,row,col);
      printf(format,x,((col==(RL-1)) ? ";" : " "));
    } printf("\n");
  }printf("];\n");
  return;
}
```

General Elementwise Matrix Operation Using Row or Column View

Using the `vsip_mrowview_f` or `vsip_mcolview_f` functions it is easy to get a vector view corresponding to any row or column of a matrix. By incrementing through all the rows or columns of matrices of the same size and using an appropriate vector elementwise function it is easy to do elementwise operations on any matrix. In example 12 below we do this in a straightforward way without manipulating any attributes. In **lines 2** through **20** we write a simple elementwise matrix add function. In **lines 22** through **40** we write a program to test the matrix add program.

Note that the matrix add function does not do any error checking and makes the assumption that the input matrices are all the same size, and that the view creates all work and have no allocation failure. To make the function more robust these errors should be checked for, and the function should return an error code instead of being void.

For this example we have arbitrarily decided to do the addition using column views. We could also have used row views. We start in **line 5** by getting the row length. Since the row length is the number of columns in the matrix this is the information necessary to calculate the number of columns we need to add over. In **lines 7** through **9** we obtain a vector view of the first column of each matrix. We then add these columns in the proper order using `vadd` in **line 10**. In **lines 11** through **16** we iterate through all the rest of the columns and add them. Note that we must destroy a view before we can reassign it, and in **lines 17** and **18** we destroy the views before returning from the function.

In main we create 3 vectors in **lines 23** through **25** and fill them with some data using `vramp` in **line 30**. These steps are just to give us some blocks with some simple to understand data in it. We don’t fill the output vector. Since we only place data in it there is no need to initialize it.

In **lines 26** through **29** we create some matrix views on the blocks created when we created the data vectors in **lines 23** through **25**. For this example we create simple row major matrices with column length of 3 and row length of 4.

The output of example 12 is below.

```
A =
[
 0.00  0.01  0.02  0.03
 0.04  0.05  0.06  0.07
 0.08  0.09  0.10  0.11
];
B =
[
 0.00  1.00  2.00  3.00
 4.00  5.00  6.00  7.00
 8.00  9.00 10.00 11.00
];
A + B = C =
[
 0.00  1.01  2.02  3.03
 4.04  5.05  6.06  7.07
 8.08  9.09 10.10 11.11
];
```

We note that using colview as the vector direction in example 12 is probably not the most efficient method since the vectors sent to the add function will not have the minimum stride. we also see that it is possible in this example to define single vectors of length 12 which included all the elements of each matrix and therefore would only require one call to the add function. In addition the view creates and destroys are very inefficient. In Example 13 we try to do a more efficient version of the matrix add function in Example 12.

Example 12

```
#include<vsip.h>
void VU_madd_f(vsip_mview_f* A,
               vsip_mview_f* B,
               vsip_mview_f* C){
  vsip_length L = vsip_mgetrowlength_f(A);
  vsip_index i;
  vsip_vview_f *a = vsip_mcolview_f(A,0),
               *b = vsip_mcolview_f(B,0),
               *c = vsip_mcolview_f(C,0);
  vsip_vadd_f(a,b,c);
  for(i=1; i<L; i++){
      vsip_vdestroy_f(a); a = vsip_mcolview_f(A,i);
      vsip_vdestroy_f(b); b = vsip_mcolview_f(B,i);
      vsip_vdestroy_f(c); c = vsip_mcolview_f(C,i);
      vsip_vadd_f(a,b,c);
  }
  vsip_vdestroy_f(a);vsip_vdestroy_f(b);
  vsip_vdestroy_f(c);
  return;
}

int main()
{   vsip_vview_f *a = vsip_vcreate_f(50,0),
                 *b = vsip_vcreate_f(50,0),
                 *c = vsip_vcreate_f(50,0);
   vsip_mview_f
      *A = vsip_mbind_f(vsip_vgetblock_f(a),0,4,3,1,4),
      *B = vsip_mbind_f(vsip_vgetblock_f(b),0,4,3,1,4),
      *C = vsip_mbind_f(vsip_vgetblock_f(c),0,4,3,1,4);
   vsip_vramp_f(0.0,.01,a); vsip_vramp_f(0.0,1.0,b);
   VU_madd_f(A,B,C);
   printf("A = \n");VU_mprintm_f("%5.2f ",A);
   printf("B = \n");VU_mprintm_f("%5.2f ",B);
   printf("A + B = C = \n");VU_mprintm_f("%5.2f ",C);
   vsip_mdestroy_f(A); vsip_mdestroy_f(B);
   vsip_mdestroy_f(C);
   vsip_valldestroy_f(a); vsip_valldestroy_f(b);
   vsip_valldestroy_f(c);
   return 0;
}
```

In example 13 we make a decision to use one of two methods to do the elementwise calculation. We use method one if the input matrices data may be represented as a single vector. For this method to work all three matrices must each be representable by a single vector. Otherwise we use method two which is the same as example 12 except we iterate the columns by resetting offset of the column vectors.

In **lines 7** and **8** we retrieve the attributes of each matrix. In **line 6** we set a check to false. In **lines 9** through **26** we find out if the matrices may all be represented using a single vector. To use a single vector all the matrices must have the same major direction, so we check this first in lines 10-13. We then find the difference for row major (column major) between the offset of the last element of a row (column) and the first element of the next row (column). By comparing this difference with the major direction stride to check for equality we set the check value to true if the matrices data may be represented by a vector.

If the check value is true we create a view along the major direction in **lines 28** through **33**. In **line 34** we find the number of elements in the matrix and reset each vector length to that value. We then add the vectors in **line 37**, destroy the vector views (**38,39**), and exit the subroutine. The matrix add is done. It is important that the initial view be at index zero along the major direction for the proper stride and offset.

If the check is false then we do the addition using column views starting at index zero (**42-43**). In **lines 45-47** we store the initial offset of these column views. We note that for index zero this is the same offset as the parent matrix. In **line 48** we add the first set of columns. In **lines 49-55** we iterate through the rest of the columns. We first reset the vector view offset to the next column by adding the row stride (**50-53**). We then add the vectors in **line 54**. After completion we are done adding so we destroy the vector views and return.

The main program starting in **line 63** tests the matrix add subroutine we have written. Note that this is basically the same main as example 12, except the attributes for the matrix bind function are set to different values, and we also do two matrix adds, once for method one and once for method two. Note that the ramp function filling the A and B matrices are set up so that the block elements are numbered 0.00 through 0.49 for A and 0 through 49 for B. This makes it easy to keep track of what portion of the block is referenced by the matrix view given a particular set of attributes in the matrix bind function. The reader is encouraged to try changing the arguments in the bind functions to become familiar with how matrices are accessed on a block given a particular view.

As a side note it would be easy to modify example 13 to do any elementwise function with two matrices as input and a matrix as output by passing in the vector elementwise function as a pointer, and replacing vector add with the function pointer.

Example 13 (Page 1 of 2)

```
#include<vsip.h>
void VU_madd_f(vsip_mview_f* A,
               vsip_mview_f* B,
               vsip_mview_f* C){
  vsip_mattr_f Aa,Ba,Ca; /* Matrix attributes */
  int c_bl = 0;
  vsip_mgetattrib_f(A,&Aa); vsip_mgetattrib_f(B,&Ba);
  vsip_mgetattrib_f(C,&Ca);
  { /* decide if it can be done with one vector */
    int check  = (Aa.row_stride < Aa.col_stride) ? 1 : 0;
        check += (Ba.row_stride < Ba.col_stride) ? 1 : 0;
        check += (Ca.row_stride < Ca.col_stride) ? 1 : 0;
    if((check != 3) && (check != 0)){ c_bl = 0;
    } else { vsip_stride A_c, B_c, C_c;
      if(check){
        A_c=Aa.col_stride-Aa.row_stride*(Aa.row_length-1);
        B_c=Ba.col_stride-Ba.row_stride*(Ba.row_length-1);
        C_c=Ca.col_stride-Ca.row_stride*(Ca.row_length-1);
        if((A_c==Aa.row_stride)&&(B_c==Ba.row_stride)
              &&(C_c == Ca.row_stride)) c_bl=1;
      }else{
        A_c=Aa.row_stride-Aa.col_stride*(Aa.col_length-1);
        B_c=Ba.row_stride-Ba.col_stride*(Ba.col_length-1);
        C_c=Ca.row_stride-Ca.col_stride*(Ca.col_length-1);
        if((A_c==Aa.col_stride) && (B_c==Ba.col_stride)
               &&(C_c==Ca.col_stride)) c_bl=1;}
    }
  }if(c_bl){ /* everything can be made into a vector */
    vsip_vview_f *a=(Aa.row_stride < Aa.col_stride) ?
          vsip_mrowview_f(A,0) : vsip_mcolview_f(A,0),
                 *b=(Ba.row_stride < Ba.col_stride) ?
          vsip_mrowview_f(B,0) : vsip_mcolview_f(B,0),
                 *c=(Ca.row_stride < Ca.col_stride) ?
          vsip_mrowview_f(C,0) : vsip_mcolview_f(C,0);
    vsip_length Nlen=Aa.row_length*Aa.col_length;
    vsip_vputlength_f(a,Nlen);vsip_vputlength_f(b,Nlen);
    vsip_vputlength_f(c,Nlen); vsip_vadd_f(a,b,c);
    vsip_vdestroy_f(a);vsip_vdestroy_f(b);
    vsip_vdestroy_f(c); printf("method 1\n");
}else{/* add by columns */
     vsip_index i;
     vsip_vview_f *a = vsip_mcolview_f(A,0),
                  *b = vsip_mcolview_f(B,0),
                   *c = vsip_mcolview_f(C,0);
```

Example 13 (Page 2 of 2

```
      vsip_offset a_o = Aa.offset,
                  b_o = Ba.offset,
                  c_o = Ca.offset;
      vsip_vadd_f(a,b,c);
      for(i=1; i<Aa.row_length; i++){
        a_o += Aa.row_stride; b_o += Ba.row_stride;
        c_o += Ca.row_stride;
        vsip_vputoffset_f(a,a_o);vsip_vputoffset_f(b,b_o);
        vsip_vputoffset_f(c,c_o);
        vsip_vadd_f(a,b,c);
      }
      vsip_vdestroy_f(a);vsip_vdestroy_f(b);
      vsip_vdestroy_f(c);
      printf("method 2\n");
   }
   return;
}
int main()
{  vsip_vview_f *a = vsip_vcreate_f(50,0),
                *b = vsip_vcreate_f(50,0),
                *c = vsip_vcreate_f(50,0);
   vsip_mview_f
     *A = vsip_mbind_f(vsip_vgetblock_f(a),3,10,3,3,4),
     *B = vsip_mbind_f(vsip_vgetblock_f(b),0,8,3,2,4),
     *C = vsip_mbind_f(vsip_vgetblock_f(c),10,4,3,1,4);
   vsip_vramp_f(0.0,.01,a); vsip_vramp_f(0.0,1.0,b);
   VU_madd_f(A,B,C);
   printf("A = \n");VU_mprintm_f("%5.2f ",A);
   printf("B = \n");VU_mprintm_f("%5.2f ",B);
   printf("A + B = C = \n");VU_mprintm_f("%5.2f ",C);
   vsip_mdestroy_f(A);vsip_mdestroy_f(B);
   vsip_mdestroy_f(C);
   A = vsip_mbind_f(vsip_vgetblock_f(a),3,8,3,2,4),
   B = vsip_mbind_f(vsip_vgetblock_f(b),0,4,3,1,4),
   C = vsip_mbind_f(vsip_vgetblock_f(c),10,4,3,1,4);
   VU_madd_f(A,B,C);
   printf("A = \n");VU_mprintm_f("%5.2f ",A);
   printf("B = \n");VU_mprintm_f("%5.2f ",B);
   printf("A + B = C = \n");VU_mprintm_f("%5.2f ",C);
   vsip_mdestroy_f(A); vsip_mdestroy_f(B);
   vsip_mdestroy_f(C);
   A = vsip_mbind_f(vsip_vgetblock_f(a),3,10,3,3,4),
   vsip_valldestroy_f(a); vsip_valldestroy_f(b);
   vsip_valldestroy_f(c);return 0;}
```

CHAPTER 5 Introduction to Vector Index Views, Boolean views, Gather, Scatter, and Indexbool

Introduction

The VSIPL specification supports vector data types which contain index values. Only vectors are supported for this purpose and there are no matrix data types which may contain index values. For the VSIPL Core profile there are vectors and matrices, so the core profile supports vector views of type vector index and vector view of type matrix index. In this chapter we will spend most of our time on vector views of type vector index which we term vector index view.

In addition the VSIPL Core profile requires support for boolean vector views. A vector boolean view may be used to create a vector index view using the **`vsip_vindexbool`** function.

Vector Index Views

By this stage in the book the reader should be fairly comfortable with manipulating vector views of type float. There are no differences with a vector index view. The data type stored in the vector index is of type **`vsip_scalar_vi`**. This type is an unsigned integer of sufficient size to index any possible vector for the implementation.

There are few elementwise functions in VSIPL Core which operate on vector index views. You may retrieve or put an index in a vector using the standard get or put, and you may copy an index using the copy functions. It is also possible to define a user data array of type vector index (**`vsip_scalar_vi index[size_of_user_data_array]`**) and then bind the user data to a block of type vector index (**`vsip_block_vi`**). Vector index views (**`vsip_vview_vi`**) may then be bound to the user block. After manipulating the user data array using normal ANSI C methods the block may be admitted to VSIPL (or released as required) and views (user data) bound to the block may be used in the normal manner.

The main method to set vector index views are to use boolean vector views created using a logical function, and then use the index boolean function to fill the index vector using the boolean vector. The index vector created may then be used to select values from a vector associated with the logical operation and place the values in another vector using a gather or scatter.

Vector Boolean Views

All of the standard vector manipulation methods supplied with the support functions, including user data arrays defined as type **`vsip_scalar_bl`** which may be used with boolean vector views using the standard VSIPL block bind, block admit and block release methods. A boolean internal to VSIPL is vendor dependent. Only the interface to boolean is defined. We will try to cover most of those interface properties here.

For get functions a zero will be returned if the indexed value is false, and a non-zero will be returned if the indexed value is true (the exact returned value for boolean get for true is vendor dependent). For put a zero is put as false and a non-zero is put as true.

For a copy of a boolean to a float vector a true is copied as 1.0 and a false as 0.0. For a copy of a float vector to a boolean a 0.0 is copied as a false and everything else is copied as true. Since it is difficult to get exactly 0.0 for float values using standard calculations users are cautioned about copying float vectors to boolean.

Any function which returns a boolean of type **`vsip_scalar_bl`** is required to test true for **`VSIP_TRUE`** and false for **`VSIP_FALSE`** using standard ANSI C tests. The values **`VSIP_TRUE`** and **`VSIP_FALSE`** are defined in the VSIP header file (**`vsip.h`**).

A first example using the scalar vector index

Before exploring vector index using vector index views lets do a simple example program using the scalar vector index. In example 14 we do a simple sort subroutine.

Note that VSIPL includes a type definition **`vsip_index`** which is the same as **`vsip_scaclar_vi`**. in **line 5** we set the index value. Many value selection functions return as an argument the index of the value selected. Since we want our sort function to start at minimum values and go to maximum we use the minimum value function to retrieve the minimum value in the array. We clone the input array in **line 4** since we want to change attributes in it. Note that "bullet proof code" would check for **`NULL`** here and return an error if there is no room to create the view. We could also save the attributes and restore them at the end. For this example the author uses individual get attribute functions, but an attribute structure could have been used as well.

The actual sorting is done in **lines 9 - 17**. The value stored at index zero is saved to a temporary storage. The minimum value is then found in **lines 11-12** and stored in the zero index location. When the minimum value is found it's original location is returned in the **`index.`** The value in the temporary location is stored in **line 13** where the minimum value was. We now have the minimum value in the array stored in location zero. We then reduce the length of the array by one, and increase the offset so that the cloned array zero location points to the second location of the original array. We do the algorithm again to find the minimum value and store it in the first location of the new array. When the length (decremented in **line 9**) reaches one we know the array has been sorted. We destroy the cloned view in **line 18** and return.

The program used to test the example in **lines 21** through **34** makes an arbitrary float user data array, binds it to a block, binds the block to a view, prints the input data, admits the block, sorts the vector view, releases the block, and finally prints the output user data. The output of **`main`** is:

```
input 5.0 -3.0  3.0 2.0  1.0 9.0 8.5 11.5  9.0
```

output -3.0 1.0 2.0 3.0 5.0 8.5 9.0 9.0 11.5

Example 14

```
#include<vsip.h>
void VU_vsort_f(vsip_vview_f *x){ /* do this in place */
  vsip_scalar_f temp; /* need to store some temp data */
  vsip_vview_f *x_clone = vsip_vcloneview_f(x);
  vsip_index index;
  vsip_length x_length = vsip_vgetlength_f(x);
  vsip_offset x_offset = vsip_vgetoffset_f(x);
  vsip_stride x_stride = vsip_vgetstride_f(x);
  while(x_length-- >1){
     temp = vsip_vget_f(x_clone,0);
     vsip_vput_f(x_clone,0,
          vsip_vminval_f(x_clone,&index));
     vsip_vput_f(x_clone,index,temp);
     x_offset += x_stride;
     vsip_vputlength_f(x_clone,x_length);
     vsip_vputoffset_f(x_clone,x_offset);
  }
  vsip_vdestroy_f(x_clone);
  return;
}
int main(){
  vsip_scalar_f u_data[] = {5.0, -3.0, 3.0, 2.0, 1.0, 9.0,
                            8.5, 11.5, 9.0};
  vsip_block_f *u_block = vsip_blockbind_f(u_data,9,0);
  vsip_vview_f  *u_view = vsip_vbind_f(u_block,0,1,9);
  printf("input\n");
  {int i; for(i=0; i<9; i++) printf("%5.1f\n",u_data[i]);}
  vsip_blockadmit_f(u_block, VSIP_TRUE);
  VU_vsort_f(u_view);
  vsip_blockrelease_f(u_block,VSIP_TRUE);
  printf("output\n");
  {int i; for(i=0; i<9; i++) printf("%5.1f\n",u_data[i]);}
  return 0;
}
```

Boolean and Vector Index Views

Example 15 demonstrates the use and creation of Boolean and Vector index views. In this example we create two vector views and determine all locations where the views are equal. In **lines 11-12** we create one ramp increasing and one ramp decreasing. They cross at zero so there is exactly one time where the two ramps are equal.

In **lines 6** and **7** we create a boolean vector and a vector index view of size equal to the size of the data vectors. In **line 19** we do a vector logical equal to compare the values of the input vectors elementwise. The output (true or false) is placed in the boolean vector **`ab_bl.`**

In **line 21** we check to see if there are any true values in the boolean output of the logical equal. If there are we do **`vsip_vindexbool`** to extract the index values of the true locations into the index vector. The index vector must be of sufficient size to hold all the possible true indices. The number of true values are returned, and the length of the index vector is reset to the actual number of indices input into it.

We finish by printing out all the indices where the elements of the input vectors to logical equal are equal.

Note that the index vector must be large enough to hold all the indices returned by **`vsip_vindexbool`**. Since the index vector length is reset to the actual number returned, then the length of the input vector index view must be reset before it is used again.

The output of example 15 follows.

```
index    A      B
  0    -2.0    2.0
  1    -1.0    1.0
  2     0.0    0.0
  3     1.0   -1.0
  4     2.0   -2.0
  5     3.0   -3.0
  6     4.0   -4.0
  7     5.0   -5.0
  8     6.0   -6.0
A = B at index   2
```

Example 15

```
#include<vsip.h>
#define L  9   /* length */
int main(){
    vsip_vview_f *a = vsip_vcreate_f(L,0),
                 *b = vsip_vcreate_f(L,0);
    vsip_vview_bl *ab_bl = vsip_vcreate_bl(L,0);
    vsip_vview_vi *ab_vi = vsip_vcreate_vi(L,0);
    vsip_length    numTrue = 0;
    int i = 0;
    /* Make up some data */
    vsip_vramp_f(-2.0, 1 , a);
    vsip_vramp_f(2.0, -1 , b);
    printf("index    A       B\n");
    for(i = 0; i<L; i++)
       printf("%3i %7.1f %7.1f \n", i,
              vsip_vget_f(a,i),
              vsip_vget_f(b,i));

    vsip_vleq_f(a,b,ab_bl);

    if(vsip_vanytrue_bl(ab_bl)){
        numTrue = vsip_vindexbool(ab_bl,ab_vi);
        for(i = 0; i < numTrue; i++)
           printf("A = B at index %3i\n",
           (int)vsip_vget_vi(ab_vi,i));
    }
    else{
        printf("No true cases\n");
    }
    vsip_valldestroy_f(a);
    vsip_valldestroy_f(b);
    vsip_valldestroy_bl(ab_bl);
    vsip_valldestroy_vi(ab_vi);
    return 0;
}
```

Gather and Scatter

To finish this chapter we demonstrate the use of gather and scatter using a couple of simple examples.

In example 16 we create a vector and find all the values greater than zero. We then place these values in a vector and print them.

In **lines 5-8** we make the data space and vectors we plan to use. In **lines12-13** we create a cosine wave for angles between zero and two π. We then fill a zero vector to compare the cosine vector with in **line 15**. Note we could use a vector with zero stride here and save some space, but we need this vector anyway to copy output data to.

In **line 16** we compare the cosine to the zero vector and output a boolean vector set to true where the cosine vector is greater than zero.

In **line 18** we recover the index values where the cosine vector is greater than zero and place them in the index vector. Note we check the return value to ensure that there were some values greater than zero.

We then use the gather function to collect the cosine values greater than zero and place them into the zero vector in **line 21**. Note that we first reset the vector length of the zero vector to be equal to the number of true values returned in **line 18** by `vsip_vindexbool`. Since the vector put length function returns a pointer to the vector view we can use it directly in the `vsip_vindexbool` function.In a gather the input vector index view is read in order. The index read is used to read the input data vector. The result is used in order and placed in the output vector. So for a gather the input vector index view and the output data view are indexed the same. The output data view length must be set to the same length as the vector index view before gather is used. The input data view is indexed using the value read from the input vector index view.

Finally we print the results, destroy the data space and exit.

For example 17 we use both gather and scatter. For this example we create a clipped cosine wave.

In example 17 **lines 5 - 26** we simply do what was done in example 16. At **line 27** we fill the original cosine view with the clip value, which for this case is zero. Then in **line 28** we replace the cosine values above the clip value using a scatter function. In a gather the input vector index view's values are used to find data in the input data vector. In a scatter the input vector index view's values are used to place data in the output vector. For the scatter case the index vector view and the input data vector are indexed the same (in order). So the data gathered in the first part of the code is now scattered back to its proper location. The result is a clipped cosine.

Note one would normally do a clip using the clip function. This example is to demonstrate scatter, not to demonstrate how to clip data.

As a final comment we note that for a scatter the index vector may contain duplicate entries. The final value stored in the output vector is vendor dependent.

Example 16

```
#include<vsip.h>
#define L 20 /* A length*/
int main()
{
    vsip_vview_f*   a = vsip_vcreate_f(L,0);
    vsip_vview_f*   b = vsip_vcreate_f(L,0);
    vsip_vview_vi*  ab_vi = vsip_vcreate_vi(L,0);
    vsip_vview_bl*  ab_bl= vsip_vcreate_bl(L,0);
    int i;
    vsip_length N;
    /* make up some data */
    vsip_vramp_f(0,2 * M_PI/(L-1),a);
    vsip_vcos_f(a,b);
    /* find out where b is greater than zero */
    vsip_vfill_f(0,a);
    vsip_vlgt_f(b,a,ab_bl);
    /* find the index where b is greater than zero */
    if((N = vsip_vindexbool(ab_bl,ab_vi))){
        /* make a vector of those points where b
           is greater than zero*/
        vsip_vgather_f(b,ab_vi,vsip_vputlength_f(a,N));
        /*print out the results */
        printf("Index   Value\n");
        for(i=0; i<N; i++)
           printf("%li      %6.3f\n",
             vsip_vget_vi(ab_vi,i),
             vsip_vget_f(a,i));
     }
     else{ printf("Zero Length Index");
     }
     vsip_valldestroy_f(a);
     vsip_valldestroy_f(b);
     vsip_valldestroy_vi(ab_vi);
     vsip_valldestroy_bl(ab_bl);
     return 0;
}
```

Example 17

```
#include<vsip.h>
#define L 50 /* A length*/
int main()
{
    vsip_vview_f *a = vsip_vcreate_f(L,0),
                 *b = vsip_vcreate_f(L,0);
    vsip_vview_vi *ab_vi = vsip_vcreate_vi(L,0);
    vsip_vview_bl *ab_bl= vsip_vcreate_bl(L,0);
    int i;
    vsip_length N;
    /* make up some data */
    vsip_vramp_f(0,2 * M_PI/(L-1),a);
    vsip_vcos_f(a,b);
    /* find out where b is greater than zero */
    vsip_vfill_f(0,a);
    vsip_vlgt_f(b,a,ab_bl);
    /* find the index where b is greater than zero */
    if((N = vsip_vindexbool(ab_bl,ab_vi))){
        /* make a vector of those points where b
           is greater than zero*/
        vsip_vgather_f(b,ab_vi,vsip_vputlength_f(a,N));
     }
     else{
        printf("Zero Length ab_vi");
        exit(0);
     }
     vsip_vfill_f(0,b);
     vsip_vscatter_f(a,b,ab_vi); /* cliped cosine */
     for(i=0; i<L; i++)
        printf("%6.3f\n",vsip_vget_f(b,i));
     /*recover the data space*/
     vsip_valldestroy_f(a);
     vsip_valldestroy_f(b);
     vsip_valldestroy_vi(ab_vi);
     vsip_valldestroy_bl(ab_bl);
     return 0;
}
```

CHAPTER 6 Signal Processing Functionality in the VSIPL Core Profile

Introduction

For the core profile signal processing functions included are, with the exception of multiple FFT, defined for use on single vectors. Multiple FFT is also one dimensional, except it is done over a matrix input. With the exception of the IIR filter routines almost all the one dimensional functionality of the VSIPL signal processing specification is contained in the core profile.

Window Creation

VSIPL provides functions to create Blackman, Chebyshev, Hanning and Kaiser windows. Unlike most functions in VSIPL the window creation routines do not use an already created vector and fill it. Instead they actually create a block, allocate data for the block, create a unit stride full length vector on the block, fill the vector with the window coefficients, and then return the pointer to the vector view. The return value will be NULL on an allocation failure, and careful programmers will check this (as the examples demonstrate the author tends to be not very careful).

The four window functions are standard and discussed in many texts. The actual formulas for the windows are included in the VSIPL standard and will not be included here. In Example 18 below we look at window creation for the Chebyshev window.

In **lines 6** and **7** we define a couple of VSIP user functions to allow us to print vectors to a file (Vector File Print y gnuplot => vfprintyg in case your wondering), and to rearrange an fft output so the DC value goes to the middle of the vector. VSIPL has defined a function to allow this, but it is not included in the core profile. These functions are in **lines 38-81**.

On **line 8-9** we calculate the Chebyshev window. In **lines 12-14** we create a complex vector for calculating the frequency response. We initialize the vector to zero. This is important if the vector contents are not replaced in some other step. For this example the real part is replaced with the window, but the imaginary part is initialized using the complex vector fill done on **line 14**.

In **line 18** we copy the window to the real portion of the complex vector. Note that frequently we can do things directly in a real view, versus doing a copy, but the window creation method in VSIPL does not allow this, since the windows data space is created directly. In **line 18** and 19 we find the fourier transform of the window and it’s magnitude squared value.

Example 18 (1 of 2)

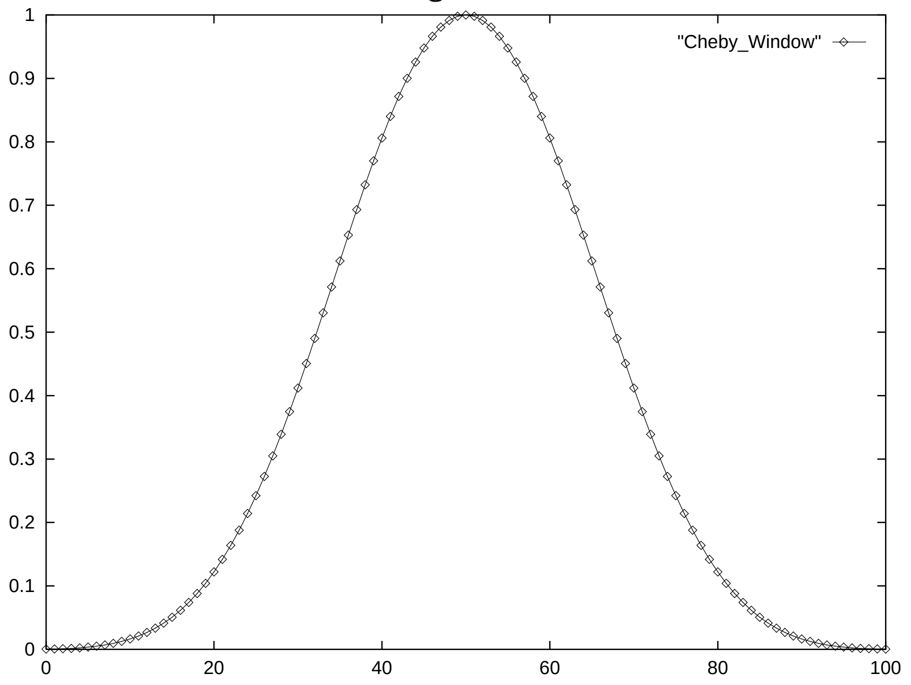
```
#include<vsip.h>
#define ripple 100  /* First side lobe 100 db down */
#define Nlength 101 /* window length */
int main()
{
     void VU_vfprintyg_f(char*,vsip_vview_f*,char*);
     void VU_vfreqswapIP_f(vsip_vview_f*);
     vsip_vview_f* Cw = vsip_vcreate_cheby_f(
               Nlength,ripple,0); /* window create here */
     vsip_fft_f *fft  = vsip_ccfftip_create_f(
                   Nlength,1.0,VSIP_FFT_FWD,0,0);
     vsip_cvview_f* FCW = vsip_cvcreate_f(Nlength,0);
     VU_vfprintyg_f("%6.8f\n",Cw,"Cheby_Window");
     vsip_cvfill_f(vsip_cmplx_f(0,0),FCW);
     { /* look at frequency response */
       vsip_vview_f *rv = vsip_vrealview_f(FCW);
       vsip_vcopy_f_f(Cw,rv);
       vsip_ccfftip_f(fft,FCW);
       vsip_vcmagsq_f(FCW,rv);
       { /* scale by 130 dB min to max*/
         vsip_index ind;
         vsip_scalar_f max = vsip_vmaxval_f(rv,&ind);
         vsip_scalar_f min = max * (1e-13);
         vsip_vclip_f(rv,min,max,min,max,rv);
       }
       vsip_vlog10_f(rv,rv);
       vsip_svmul_f(10,rv,rv);
       VU_vfreqswapIP_f(rv);
       VU_vfprintyg_f("%6.8f\n",rv,
           " Cheby_Window_Frequency_Response");
       vsip_vdestroy_f(rv);
     }
     vsip_fft_destroy_f(fft);
     vsip_valldestroy_f(Cw);
     vsip_cvalldestroy_f(FCW);
     return 0;
}
```

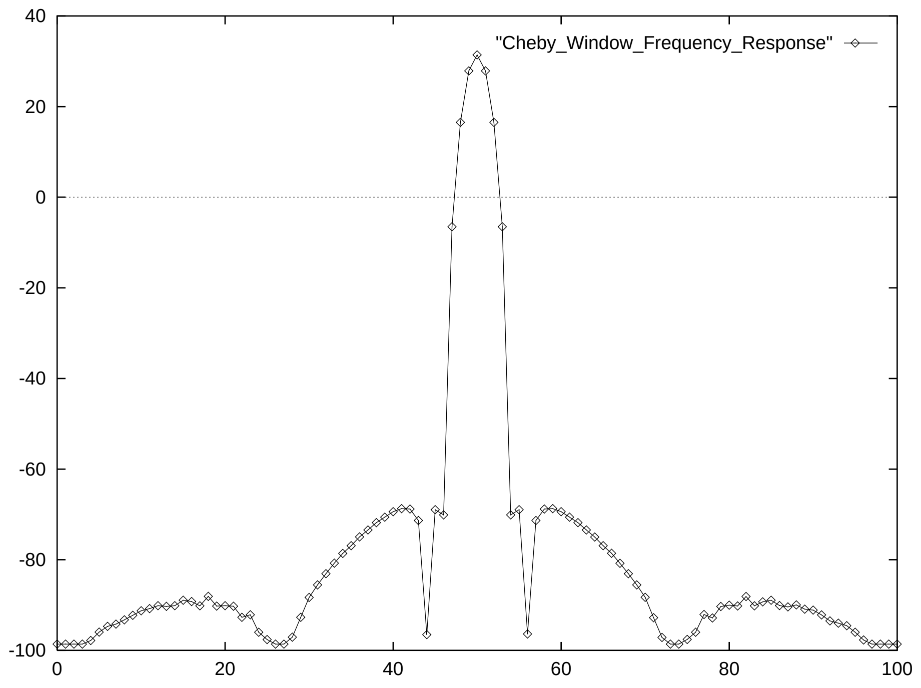
Example 18 (2 of 2)

```
void VU_vfreqswapIP_f(vsip_vview_f* b)
{   vsip_length N = vsip_vgetlength_f(b);
    if(N%2){/* odd */
       vsip_vview_f *a1 = vsip_vsubview_f(b,
                 (vsip_index)(N/2)+1,
                 (vsip_length)(N/2));
       vsip_vview_f *a2 = vsip_vsubview_f(b,
                 (vsip_index)0,
                 (vsip_length)(N/2)+1);
       vsip_vview_f *a3 = vsip_vcreate_f(
                 (vsip_length)(N/2)+1,
                  VSIP_MEM_NONE);
       vsip_vcopy_f_f(a2,a3);
       vsip_vputlength_f(a2,(vsip_length)(N/2));
       vsip_vcopy_f_f(a1,a2);
       vsip_vputlength_f(a2,(vsip_length)(N/2) + 1);
       vsip_vputoffset_f(a2,(vsip_offset)(N/2));
       vsip_vcopy_f_f(a3,a2);
       vsip_vdestroy_f(a1); vsip_vdestroy_f(a2);
       vsip_valldestroy_f(a3);
    }else{ /* even */
       vsip_vview_f *a1 = vsip_vsubview_f(b,
                 (vsip_index)(N/2),
                 (vsip_length)(N/2));
       vsip_vputlength_f(b,(vsip_length)(N/2));
       vsip_vswap_f(b,a1);
       vsip_vdestroy_f(a1);
       vsip_vputlength_f(b,N);
    }
    return;
}


void VU_vfprintyg_f(char* format,
                    vsip_vview_f* a,
                    char* fname)
{   vsip_length N = vsip_vgetlength_f(a);
    vsip_length i;
    FILE *of = fopen(fname,"w");
    for(i=0; i<N; i++)
           fprintf(of,format, vsip_vget_f(a,i));
    fclose(of);
    return;
}
```

Figure 2





Output of example 18 plotted using gnuplot. The top plot is the Chebyshev window, the bottom the corresponding frequency response.

For the window in example 18 we requested 100 db between the highest sidelobe and the main lobe; however the minimum value could be zero. Very small or zero values are inconvenient when doing logs. In **lines 20-25** we scale the frequency response values to 130 dB. We find 10 log base 10 of the values in **lines 26-27** (this is the dB value), and in **line 29-30** print them out. Figure 2 is the resultant window, and it's frequency response.

The author notes that he has run example 18 on two platforms, a Sun server and a Pentium based Linux computer, but for some reason the result of the float version of the Chebyshev window on the Sun is not very good (the first side lobe is only down about 60 dB on the Sun float version). The double version on the Sun platform works fine. The version in figure 2 is the float version from the pentium platform. The library and example code is identical on both platforms. The author has not tracked down the problem yet.

Convolution, Correlation and FIR Filtering

There are many similarities between these three functions. The convolution and FIR use a kernel vector in there object creation which is stored in the object; however the correlation uses a reference data vector which is not stored in the object. The convolution and FIR are basically filter functions and allow de-sampling. The correlation has an option to remove bias. The correlation and convolution have options to indicate what portion of the output is desired. The convolution is designed to filter a single piece of data, the FIR object maintains state information so that it may be used to filter a continuous data stream.

The FIR has already been demonstrated in example 11. In this section we will do two additional examples, one for convolution and one for correlation. In the convolution example the FIR and convolution results for identical kernels will be compared.

Correlation

The Correlation example is number 19. In **lines 2-3** we define a frequency of 5 Hz for creating data and a sample rate of 128 Hz. In **lines 4-5** we define **`Nval`** as 75 (the length of the input vector) and **`Mval`** as 51 (the length of the reference vector). In **lines 11-16** we create some data space. Note that **`z_y`** is used to store lag time values, and **`z_xh`** is used to store sample time values. These allow for nicer plotting, but are not really necessary. Note that the function in **line 7** (code in **lines 68-80**) allows easy printing to a file of x, y values suitable for use in a simple plotting package.

In **line 17** we create a correlation object for *full* support (**`VSIP_SUPPORT_FULL`**). In **line 19** we create a ramp for use in creating our input vector and reference vector in **lines 25-26** and **23-24** respectively. Note that the reference vector length must be less than or equal to the input vector length. In **lines 20-22** we set up the time vector and the lag vector.

In **lines 32** through **43** we do a biased correlation and save it in file "*y_full*" and an unbiased correlation and save it in file "*y_full_unbiased*". In **line 43** we destroy the correlation object so we can reuse it. We then create and do a biased correlation for *same* support (VSIP_SUPPORT_SAME) in **lines 44-51** storing the result in file "*y_same*", and in **lines 53** through **60** we create and do a biased correlation for *minimum* support storing the result in file “*y_min*”. Note that we must destroy a correlation object before creating a new one (**lines 51** and **52**) for the same reason we must destroy a vector object before creating a new one.

In example 19 we have not done every possible case for unbiased but the results can be inferred from the biased case. The results of example 19 are plotted in figure 3.

Note that for the minimum support case a point is only output for correlation where every point of the reference vector is used in the calculation. If the reference vector and the input vector are both the same length only one point will be output for the minimum support case.

In the correlation plot of figure 3 for the unbiased case the three supports are each plotted in a separate color, black for full support, blue for same support, and red for minimum support.

Example 19 (1 of 2)

```
#include<vsip.h>
#define f0 5      /* Frequency */
#define fs 128    /* Sample rate */
#define Nval 75   /* Input Data Length */
#define Mval 51   /* Reference Data Length */

void VU_vfprintxyg_f(
       char*,vsip_vview_f*,vsip_vview_f*,char*);
int main()
{
   vsip_length max_l = Nval + Mval - 1;
   vsip_vview_f *x = vsip_vcreate_f(Nval,0);
   vsip_vview_f *h = vsip_vcreate_f(Mval,0);
   vsip_vview_f *y = vsip_vcreate_f(max_l,0);
   vsip_vview_f *z_y = vsip_vcreate_f(max_l,0);
   vsip_vview_f *z_xh = vsip_vcreate_f(Nval,0);
   vsip_corr1d_f *cor = vsip_corr1d_create_f(
              Mval,Nval,VSIP_SUPPORT_FULL,0,0);
   vsip_vramp_f(0,(2 * M_PI * f0)/ fs,x);
   vsip_vramp_f(-((vsip_scalar_f)max_l)/(2.0 * fs),
              1.0/(fs),z_y);
   vsip_vramp_f(0.0,1.0/(fs),z_xh);
   vsip_vputlength_f(x,Mval);
   vsip_vsin_f(x,h);
   vsip_vputlength_f(x,Nval);
   vsip_vsin_f(x,x);
   vsip_vputlength_f(z_xh,Mval /*h length*/);
   VU_vfprintxyg_f("%8.6f %8.6f\n",z_xh,h,"h_data");
   vsip_vputlength_f(z_xh,Nval /*x length*/);
   VU_vfprintxyg_f("%8.6f %8.6f\n",z_xh,x,"x_data");

   vsip_correlate1d_f(cor,VSIP_BIASED,h,x,y);
   vsip_vputoffset_f(z_y,0);
   vsip_vputlength_f(z_y,vsip_vgetlength_f(y));
   VU_vfprintxyg_f("%8.6f %8.6f\n",z_y,y,"y_full");

   vsip_correlate1d_f(cor,VSIP_UNBIASED,h,x,y);
   VU_vfprintxyg_f(
              "%8.6f %8.6f\n",z_y,y,"y_full_unbiased");
   vsip_corr1d_destroy_f(cor);
```

Example 19 (2 of 2)

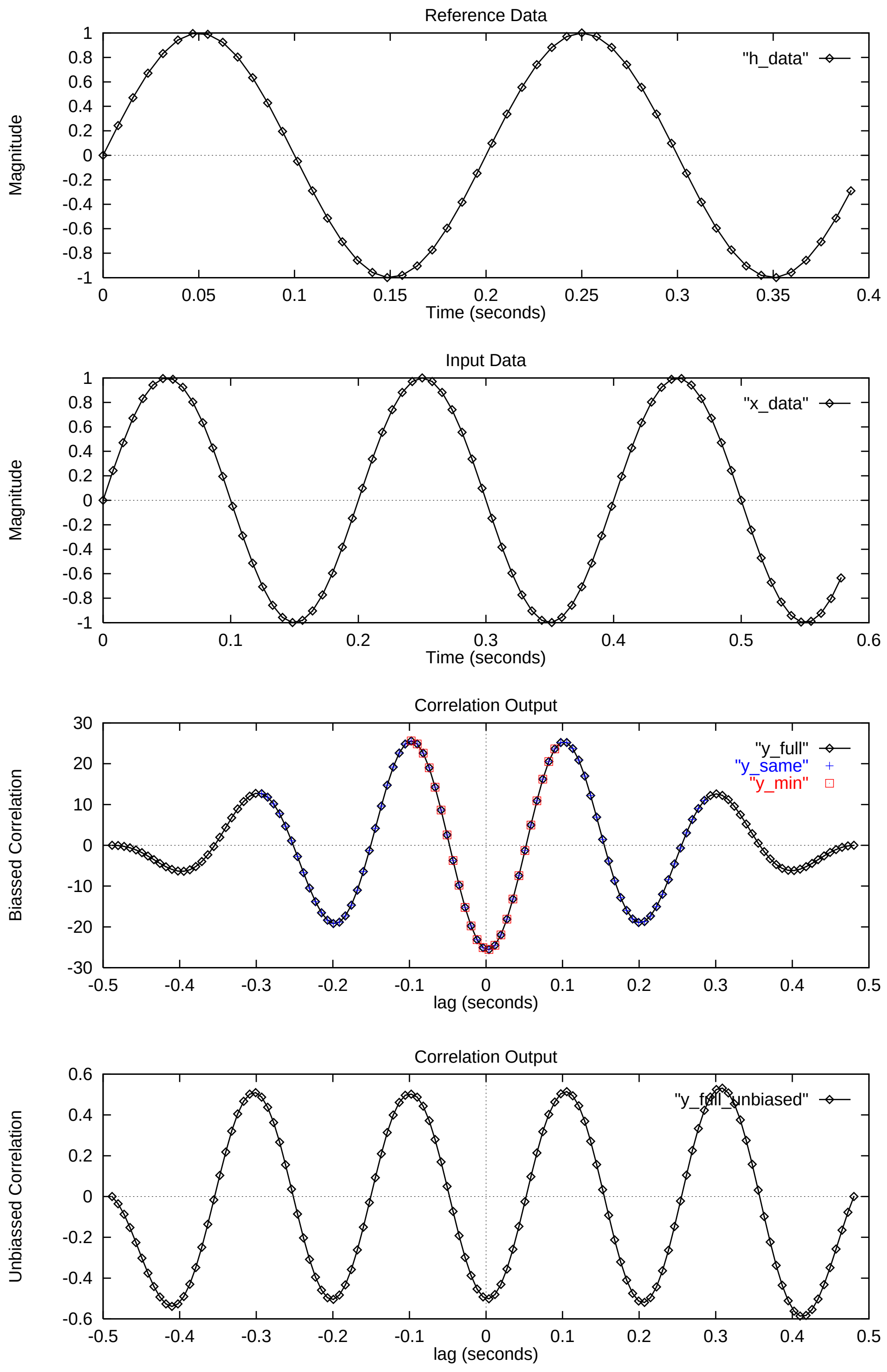
```
   cor = vsip_corr1d_create_f(
                Mval,Nval,VSIP_SUPPORT_SAME,0,0);
   vsip_vputlength_f(y,Nval);
   vsip_vputlength_f(z_y,Nval);
   vsip_vputoffset_f(z_y,(vsip_offset)(Mval/2));
   vsip_correlate1d_f(cor,VSIP_BIASED,h,x,y);
   VU_vfprintxyg_f("%8.6f %8.6f\n",z_y,y,"y_same");
   vsip_corr1d_destroy_f(cor);

   cor = vsip_corr1d_create_f(
                Mval,Nval,VSIP_SUPPORT_MIN,0,0);
   vsip_vputlength_f(y,Nval-Mval+1);
   vsip_vputoffset_f(z_y,(vsip_offset)(Mval - 1));
   vsip_vputlength_f(z_y,Nval-Mval+1);
   vsip_correlate1d_f(cor,VSIP_BIASED,h,x,y);
   VU_vfprintxyg_f("%8.6f %8.6f\n",z_y,y,"y_min");
   vsip_corr1d_destroy_f(cor);

   vsip_valldestroy_f(x);
   vsip_valldestroy_f(h);
   vsip_valldestroy_f(y);
   return 0;
}

void VU_vfprintxyg_f(char* format,
                     vsip_vview_f* x,
                     vsip_vview_f* y,
                     char* fname){
    vsip_length N = vsip_vgetlength_f(y);
    vsip_length i;
    FILE *of = fopen(fname,"w");
    for(i=0; i<N; i++)
      fprintf(of,format,
               vsip_vget_f(x,i), vsip_vget_f(y,i));
    fclose(of);
    return;
}
```

Figure 3



Output of example 19 plotted using gnuplot. Note that the input data and the reference data are not the same number of samples, although they are the same sample rate and frequency.

Convolution

The convolution is very similar to the FIR filter, except the FIR is designed to be done on continuous data, and the Convolution is designed to be done on a single data vector. In addition since the convolution is done only once the portion of the output (support type definition `vsip_support_region`) is defined for minimum, same or full output similar to the correlation function. The outputs for the convolution and the FIR for the same kernel should be similar. In example 20 we create a Kaiser window in **line 9-10** and use this as the kernel for use in the convolution creates and FIR creates. We use the non-portable random number generator to create $N(0, 1)$ gaussian data for an input. We do two cases, one with a decimation factor of one (no decimation) and one with a decimation factor of three. The output is displayed in figure 4.

We note that the current version of TASP_VSIPL uses a time domain moving weighted (by the kernel) sum with save information between data sets to calculate the output. For the FIR decimation is done on the fly by skipping calculations not needed in the output. The convolution uses an FFT method and does the decimation only on the final output, after the total convolution is complete. The two methods should be equivalent, with some small calculation errors. The output shows up a larger sidelobe level (see figure 4) for the FIR method with decimation. The author is not sure what is going on here.

We want to display the results as a frequency response in dB. The routine `VU_vfrdB_f` in **lines 134-177** computes a simple frequency response. Note the routine returns one on allocation failure and a zero on success. We don’t use this feature in the main routine, but it is handy to be able to check for allocation failures. We also allow for a range to be passed into the routine for scaling. Even though we have a complex magnitude squared function (**line 149**) which ensures a positive result we may still have a zero in the data and log functions do not handle zero gracefully. The input is replaced with the output in this function. Although the input is always real we use a complex to complex FFT to calculate the frequency response. This is because the real to complex FFT requires an even number of input values, and we want this function to be more general than that.

A print function (in **lines 125-133**) supports output to a file suitable for use in a simple graph program.

We create the convolution and FIR objects in **lines 26-30** for the decimation factor of one case. These are destroyed in **lines 63-64** and new ones created for the decimation factor of three case in **lines 65-70**. Note that just like vectors we must destroy these objects before reassigning the pointer to new ones to avoid memory leaks.

The length of the output vector for the full support on the convolution is $\text{floor}[(N-1)/D]+1$ where N is the input data length and D is the decimation factor. The number of samples for the FIR output may not always be the same, depending on the input vector length, the decimation factor, and the state of the FIR object. The maximum output length for the FIR will be the $\text{ceiling}[N/D]$. The FIR returns a value equal to the number of new samples actually returned in the output vector. For this example we have ignored this factor since it is only one sample point at the end of the vector and only affects the decimation 3 case. The vector lengths are set in **lines 22-25** and **lines 71- 74**. The calculation for the convolution returns the correct value for maximum output length for the FIR.

We output the data for the various pieces in **lines 38-44**, **56-62**, **82-89**, **97-103**. Note the x axis is a frequency which is basically a percent of the sample rate. The largest frequency is one half the sample rate.

We output the kernel, and the kernels frequency response in **lines 105-115**.

Example 20 (1 of 4)

```
#include<vsip.h>
#define N_data 4096
#define dec1   1
#define dec3   3
int  VU_vfrdB_f(vsip_vview_f*,vsip_scalar_f );
void VU_vfprintxyg_f(char*, vsip_vview_f*,
                     vsip_vview_f*, char*);
int main () /* Start of main program******************/
{ vsip_vview_f *kernel =
        vsip_vcreate_kaiser_f(128,15.0,VSIP_MEM_NONE);
  vsip_randstate *r_state  =
        vsip_randcreate(11,1,1,VSIP_NPRNG);
  vsip_conv1d_f *conv;
  vsip_fir_f    *fir;
  vsip_vview_f *data  = vsip_vcreate_f(
        N_data,VSIP_MEM_NONE),
               *noise = vsip_vcreate_f(
        N_data,VSIP_MEM_NONE),
               *avg   = vsip_vcreate_f(
        N_data,VSIP_MEM_NONE);
  int i; vsip_length N_len;
  vsip_vputlength_f(data,
        (vsip_length)((N_data-1)/dec1)+1);
  vsip_vputlength_f(avg,
        (vsip_length)((N_data-1)/dec1)+1);
  conv = vsip_conv1d_create_f(
        kernel,VSIP_NONSYM,
        N_data,dec1,VSIP_SUPPORT_SAME,0,0);
  fir  = vsip_fir_create_f(kernel,VSIP_NONSYM,N_data,
        dec1,VSIP_STATE_SAVE,0,0);
  vsip_vfill_f(0,avg);
  for(i=0; i<10; i++){
         vsip_vrandn_f(r_state,noise);
         vsip_convolve1d_f(conv,noise,data);
         VU_vfrdB_f(data,1e-13);
         vsip_vsma_f(data,0.1,avg,avg);
  }
  N_len = vsip_vgetlength_f(avg);
  {  vsip_vview_f *x = vsip_vcreate_f(
           N_len,VSIP_MEM_NONE);
     vsip_vramp_f(-.5,1.0/(vsip_scalar_f)(N_len-1),x);
     VU_vfprintxyg_f("%8.6f %8.6f\n",x,avg,"conv_dec1");
     vsip_vdestroy_f(x);
  }
```

Example 20 (2 of 4)

```
   vsip_vfill_f(0,avg);
   for(i=0; i<10; i++){
      vsip_vrandn_f(r_state,noise);
      vsip_firflt_f(fir,noise,data);
      VU_vfrdB_f(data,1e-13);
      vsip_vsma_f(data,0.1,avg,avg);
    }
    N_len = vsip_vgetlength_f(avg);
    {  vsip_vview_f *x = vsip_vcreate_f(
             N_len,VSIP_MEM_NONE);
       vsip_vramp_f(-.5,1.0/(vsip_scalar_f)(N_len-1),x);
       VU_vfprintxyg_f("%8.6f %8.6f\n",x,avg,"fir_dec1");
       vsip_vdestroy_f(x);
    }
    vsip_conv1d_destroy_f(conv);
    vsip_fir_destroy_f(fir);
    conv = vsip_conv1d_create_f(
            kernel,VSIP_NONSYM,
            N_data,dec3,VSIP_SUPPORT_SAME,0,0);
    fir  = vsip_fir_create_f(
            kernel,VSIP_NONSYM,
            N_data,dec3,VSIP_STATE_SAVE,0,0);
    vsip_vputlength_f(data,
            (vsip_length)((N_data-1)/dec3)+1);
    vsip_vputlength_f(avg,
            (vsip_length)((N_data-1)/dec3)+1);
    vsip_vfill_f(0,avg);
    for(i=0; i<10; i++){
       vsip_vrandn_f(r_state,noise);
       vsip_convolve1d_f(conv,noise,data);
       VU_vfrdB_f(data,1e-13);
       vsip_vsma_f(data,0.1,avg,avg);
    }
    N_len = vsip_vgetlength_f(avg);
    {  vsip_vview_f *x = vsip_vcreate_f(
             N_len,VSIP_MEM_NONE);
       vsip_vramp_f(-.5,1.0/(vsip_scalar_f)(N_len - 1),x);
       VU_vfprintxyg_f("%8.6f %8.6f\n",
             x, avg,"conv_dec3");
       vsip_vdestroy_f(x);
    }
```

Example 20 (3 of 4)

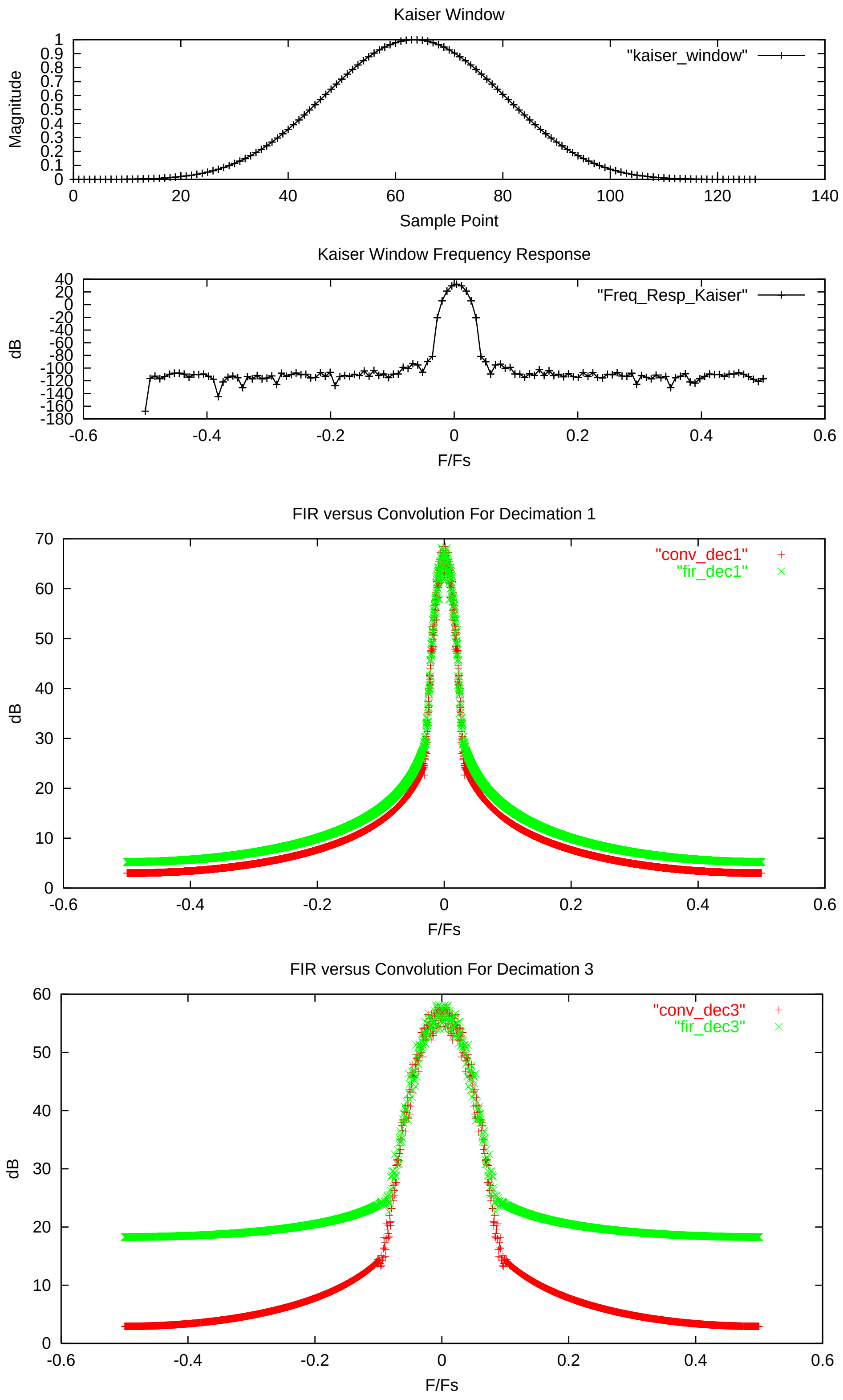
```
    vsip_vfill_f(0,avg);
    for(i=0; i<10; i++){
        vsip_vrandn_f(r_state,noise);
        vsip_firflt_f(fir,noise,data);
        VU_vfrdB_f(data,1e-13);
        vsip_vsma_f(data,0.1,avg,avg);
     }
     N_len = vsip_vgetlength_f(avg);
     {  vsip_vview_f *x = vsip_vcreate_f(
                N_len,VSIP_MEM_NONE);
        vsip_vramp_f(-.5,1.0/(vsip_scalar_f)(N_len-1),x);
        VU_vfprintxyg_f("%8.6f %8.6f\n",
            x, avg,"fir_dec3");
        vsip_vdestroy_f(x);
     }
     N_len = vsip_vgetlength_f(kernel);
     { vsip_vview_f *x = vsip_vcreate_f(
               N_len,VSIP_MEM_NONE);
       vsip_vramp_f(0,1,x);
       VU_vfprintxyg_f("%8.6f %8.6f\n",
             x,kernel,"kaiser_window");
       vsip_vramp_f(-.5,1.0/(vsip_scalar_f)(N_len-1),x);
       VU_vfrdB_f(kernel,1e-20);
       VU_vfprintxyg_f("%8.6f %8.6f\n",
             x,kernel,"Freq_Resp_Kaiser");
       vsip_vdestroy_f(x);
     }
  vsip_randdestroy(r_state);
  vsip_valldestroy_f(kernel);
  vsip_conv1d_destroy_f(conv);vsip_fir_destroy_f(fir);
  vsip_valldestroy_f(data); vsip_valldestroy_f(noise);
  vsip_valldestroy_f(avg);
  return 0;
}/*end of main program ********************************/

void VU_vfprintxyg_f(char* format,vsip_vview_f* x,
                    vsip_vview_f* y,char* fname)
{   vsip_length N = vsip_vgetlength_f(y);
    vsip_length i;
    FILE *of = fopen(fname,"w");
    for(i=0; i<N; i++)fprintf(of,
    format, vsip_vget_f(x,i),vsip_vget_f(y,i));
    fclose(of); return;
}
```

Example 20 (4 of 4)

```
int VU_vfrdB_f(vsip_vview_f *a,vsip_scalar_f range)
{ int ret = 0;
  vsip_length N_len=vsip_vgetlength_f(a);
  vsip_cvview_f *ca=vsip_cvcreate_f(N_len,VSIP_MEM_NONE);
  vsip_fft_f *fft =   vsip_ccfftip_create_f(
       N_len,1,VSIP_FFT_FWD,0,0);
  vsip_vview_f *ra = vsip_vrealview_f(ca),
               *ia = vsip_vimagview_f(ca),
               *ta = vsip_vcloneview_f(a);
  vsip_offset s = (vsip_offset)vsip_vgetstride_f(ta);
  if((ca == NULL) || (fft == NULL) || (ra == NULL) ||
     (ia == NULL) || (ta == NULL)){ret =  1;
  }else{
     vsip_vfill_f(0,ia); vsip_vcopy_f_f(a,ra);
     vsip_ccfftip_f(fft,ca);
     vsip_vcmagsq_f(ca,ra);
     {  vsip_index ind;/* scale by "range" min to max*/
        vsip_scalar_f max = vsip_vmaxval_f(ra,&ind);
        vsip_scalar_f min = max * range;
        vsip_vclip_f(ra,min,max,min,max,ra);
     }
     if(N_len%2){vsip_length Nlen = N_len/2;
         vsip_vputlength_f(ta,Nlen+1);
         vsip_vputlength_f(ra,Nlen+1);
         vsip_vputoffset_f(ta,Nlen * s);
         vsip_vcopy_f_f(ra,ta);
         vsip_vputlength_f(ra,Nlen);
         vsip_vputlength_f(ta,Nlen);
         vsip_vputoffset_f(ta,vsip_vgetoffset_f(a));
         vsip_vputoffset_f(ra,Nlen+1);
         vsip_vcopy_f_f(ra,ta);
     }else{vsip_length Nlen = N_len/2;
         vsip_vcopy_f_f(ra,ta);
         vsip_vputlength_f(ta,Nlen);
         vsip_vputlength_f(a,Nlen);
         vsip_vputoffset_f(ta,(vsip_offset)(Nlen) * s);
         vsip_vswap_f(ta,a);
         vsip_vputlength_f(a,N_len);
      }vsip_vlog10_f(a,a);vsip_svmul_f(10,a,a);
   }vsip_fft_destroy_f(fft);
     vsip_vdestroy_f(ra); vsip_vdestroy_f(ia);
     vsip_cvalldestroy_f(ca);vsip_vdestroy_f(ta);
     return ret;
}
```

Figure 4



Output of example 20. Comparison of FIR and Convolution results.

Fourier Transforms

We have used the FFT routine in several previous examples in this document. The main differences in FFT functionality between the core lite profile and the core profile is the addition of real to complex and complex to real FFTs and the addition of multiple FFTs. In example 21 we demonstrate the real to complex multiple FFT and the complex to complex multiple FFT in the computation of a wavenumber frequency plot.

Wavenumber/Frequency plot

For example 21we need a little background for the example to make sense. If the reader is already familiar with $k\omega$ plots he should skip this section. There is no VSIPL information here.

A wavenumber is $(2\pi)/\lambda$ where λ is the wavelength, and a wave vector $\vec{k}$ is the wavenumber times a unit vector in the direction of propagation. Please refer to Figure 5.

Basically we define a two dimensional medium and propagate a plan wave through it. The plane wave propagates at a speed of c. We sample the plane wave in time with sample rate f_s and in space with sample rate F_s. Note that f_s is samples per second and F_s is samples per meter. For example if our sensors were placed at one meter intervals then the sample rate in space would be one sample per meter for a wave vector parallel to the array. For the array of sensors shown in Figure 5 we can see that $\vec{k} \cdot \vec{r} = ((2\pi)/\lambda)\cos(\theta)x$ where x is the position of the sensor along the direction of the unit vector $\hat{e}_x$. Since we can place our origin any place we want we make the y component go away by setting it to zero. We now see that if we calculate the FFT of $p(t, x)$ with time (seconds) that we transform to frequency space f cycles per second (Hz), and if we do an FFT with x (meters) we transform to frequency space with frequency $((2\pi)/\lambda)\cos(\theta)$ cycles per meter. Since the equation $p(t, x) = p_o p_1(t) p_2(x)$ we can do a discrete transform in time and space independently. Note that our sensors need to be equally spaced to use the FFT method in space. Another way to say this is that F_s is a constant.

The above explanation is pretty short. For a more thorough description of frequency domain beamforming the author recommends “Array Signal Processing: Concepts and Techniques” by Johnson and Dudgeon.

Demonstration for $k\omega$

Although the purpose of example 21 is to demonstrate frequency domain beamforming using multiple fourier transform functions available in VSIPL most of the example calculations are to simulate data to beam form, and then format the output for plotting. The output is in figure 6 and was done using the “tv” plotting tool in PV-Wave. The author was not successful in finding a better way to do this type of plot using public domain tools. We will go through the code in some detail, but all the beamforming takes place in **lines 101-102** where the data is windowed (data taper) to reduce side lobes, and in **lines 105-106** where FFT beamforming is done.

In **lines 3-16** we define some constants. Our array has a sensor spacing D of 1.5, and each sensor is sampled in time at $\mathrm{Fs} = 1000\ \mathrm{Hz}$. We define some frequencies in **line 5-8** to simulate a narrow band source. The source is located on beam 30. The number of samples collected on each sensor before processing the array is $\mathrm{Ns} = 512$. We need to define some noise coming from various directions. We define a noise length $\mathrm{Nn} = 1024$ to allow for simple beamforming of the noise. The number of independent noise sources are $\mathrm{Nnoise} = 64$. We define some constants for a Kaiser window in **lines 15-16**. The Kaiser window we use as a low pass filter for the noise estimate. (The author mostly guessed around till he found some values that seemed to give a suitable output for Figure 6. There is no science to the selection of this window.) The number of sensors in our array will be $\mathrm{Mp} = 128$ and the propagation speed of the medium will be $\mathrm{c} = 1500$. Note that the propagation speed of the medium and the sensor spacing must have conformant units (meters, feet, yards) but for this example the actual length unit does not matter.

In **lines 22-23** we create windows to do the data taper in time (`windowt`) and space (`windowp`) In **lines 24-25** we create the FIR filter object. Most of the stuff through **lines 44** create various works spaces which the author may discuss latter in the program. The author is not sure he did the routine in the most efficient way so some of the work space may seem a little redundant. Note **line 29** where we create an array to hold the noise vectors. We could have kept the noise vectors in a matrix, but it would not have been as handy to work with. In **line 42** we create a constant which is basically the travel time of a signal between two sensors if the signal arrives at endfire. (The term “endfire” indicates a plain wave traveling with a wave vector parallel to the array. This is the maximum travel time between sensors.) This is normalized by the sample rate.

In **line 45** we create a state object for a portable random number generator. **Lines 47-50** create radian frequencies of our target frequencies and normalize them by the sample rate. **Lines 51** and **52** are used in the program (**lines 91-96**) for calculating the noise on each sensor for each noise direction. **Line 51** is basically a constant angular distribution for the noise from endfire to endfire, and **line 52** is an offset into a noise vector of a size great enough that the time represented by the sample at the offset is at least as large as the time for a wave to traverse the entire array. This offset is adjusted, plus or minus, to account for the travel time for the noise from a particular direction and on a particular sensor.

In **lines 58-61** we basically calculate a matrix of (normalized) time delays between the first sensor and any other sensor for all the beams we will calculate in the FFT beamformer. (We don’t actually need all these for this example, but the author did this example from another example we don’t cover here, and did not want to change this step).

We call the output matrix `gram`. In **lines 62-66** we initialize the output data matrix to zero using a rowview create-destroy cycle with a vector fill. (This is not the most efficient way to do it.)

In **lines 67-73** we fill our noise vectors for each noise direction. In **line 72** we set the length to our data length of $\text{Ns} = 512$, and in **line 71** we scale the Noise. Note that there is nothing special in the scaling. The author just tried some scaling until he got a noise level he liked.

In **line 74** we fill a normalized time vector. The actual time would require a scaling by the inverse of the sample rate, but we have done that scaling elsewhere.

Finally we are ready to fill our input matrix with (simulated) data in **lines 75-99**. We must sample our data at a different time for each sensor so we loop through each sensor (**line 75**) and select the proper time delay between sensors for beam **`Theta_o`** $= 30$ in **line 76**. We then calculate the narrow band time series and place them in the input data in **lines 78-90**.

In **lines 91-96** we estimate the proper offset in the noise vector for all the noise directions for the particular phone we are simulating data for. We do this in the time domain so it is not exact as the proper delay may be between samples. For noise we don't need to be exact. The main purpose of adding the noise is to reproduce the characteristic wedge shape of a $k\omega$ plot.

All the steps above have been to create some artificial time series data to do the frequency domain beamforming on. We now have a matrix of data.

In **lines 101-102** we window the data to reduce the sidelobes. Note we use an elementwise vector matrix multiply, first along the row for the time data, then along the column for the space data. This function does a vector elementwise multiply to each row or column of a matrix.

In **line 105** we finally get to the first multiple FFT. Since we have a matrix of time series it is faster to use the multiple FFT than to do each phone separately. We also have a real array and only need the positive portion of the frequency domain output so the function we use is the real to complex FFT (**`vsip_rcfftmop_f`**). Note that this function must be done out of place.

In **line 102** we do an FFT along the array. For this function we must use the complex to complex multiple FFT and we do it in place (**`vsip_ccfftmip_f`**). We now have transformed our time and space data to frequency direction data.

The rest of the example is spent transforming our output to be suitable for plotting. The author decided to do this with a gray scale between a minimum of 0 and a maximum of 255. We basically find the magnitude squared value of each element of the data in **line 115**, scale this output in **line 118** so the minimum value is 1.0 (log of one is zero), then take its log in **line 120**, then scale the log data to between 0 and 255 for plotting**.** In **line 126-129** we move the origin of our plot so that broadside beams (corresponding to zero (space) frequency) arrive at the middle in figure 6.

We now back up a little and look at **lines 109-113**. Here the author has used knowledge of the input matrix views attributes to extract a unit stride vector which covers the entire data space of the matrix. Since the core profile does not have elementwise matrix operations this step is needed so that the scaling in the paragraph above may be easily done using vector elementwise operations.

Also in **lines 126-129** where the origin of the space FFT is moved to the center the author has used knowledge that the number of sensors in the array are even. This algorithm will not work if there are an odd number of sensors.

Finally we print the results in **line 135**.

We note that there are many objects which are not destroyed in this program (mostly because the author got a little lazy). For instance the window vectors for the data taper and the FFT objects are not destroyed. All the memory for these objects are recovered when the program ends. Since the objects are only created once then they are not a problem; however objects internal to loops, such as data_v which is a view created on **line 77,** must be destroyed (on **line 98**). Any object which will be created more than once should be destroyed before it is recreated. Memory limited systems should also destroy objects that are no longer needed, and can't be reused.

Example 21 (1 of 4)

```
#include <vsip.h>

#define D 1.5      /* sensor spacing */
#define Fs 1000    /* sample rate Hz */
#define F0 450     /* some frequencies for a target */
#define F1 300
#define F2 150
#define F3 50
#define Theta_o 30 /* beam number of narrow band tones */
#define Ns 512     /* samples in a time series */
#define Nn 1024    /* sample in a noise series */
#define Mp 128     /* sensors in linear array */
#define c 1500     /* propagation speed */
#define Nnoise 64  /* number of noise directions */
#define kaiser 9 /* window parameter */
#define Nfilter 10 /* kernel length for noise filter */

void VU_mprintgram_f(vsip_mview_f*,char*);

int main()
{  int i,j; /* counters */
   vsip_vview_f *windowt = vsip_vcreate_hanning_f(Ns,0);
   vsip_vview_f *windowp = vsip_vcreate_hanning_f(Mp,0);
   vsip_vview_f *kernel =
         vsip_vcreate_kaiser_f(Nfilter,kaiser,0);
   vsip_fir_f *fir = vsip_fir_create_f(kernel,VSIP_NONSYM,
         2 * Nn,2,VSIP_STATE_SAVE,0,0);
   vsip_vview_f *t =vsip_vcreate_f(Ns,0); /*time vector*/
   vsip_vview_f *noise[Nnoise];
   vsip_vview_f *nv = vsip_vcreate_f(2 * Nn,0);
   vsip_vview_f *tt = vsip_vcreate_f(Ns,0);
   vsip_mview_f *data = vsip_mcreate_f(Mp,Ns,VSIP_ROW,0),
                *rmview;
   vsip_vview_f *data_v, *gram_v;
   vsip_cvview_f *gram_data_v;
   vsip_cmview_f *gram_data =
        vsip_cmcreate_f(Mp,Ns/2 + 1,VSIP_COL,0);
   vsip_mview_f  *gram =
         vsip_mcreate_f(Mp,Ns/2 + 1,VSIP_ROW,0);
   vsip_mview_f  *Xim =
         vsip_mcreate_f(Mp,Mp+1,VSIP_ROW,0);
   vsip_scalar_f alpha = (D * Fs) / c;
```

Example 21 (2 of 4)

```
    vsip_vview_f *m = vsip_vcreate_f(Mp,0);
    vsip_vview_f *Xi = vsip_vcreate_f(Mp + 1,0);
    vsip_randstate *state =
           vsip_randcreate(15,1,1,VSIP_PRNG);
    vsip_scalar_f w0 = 2 * M_PI * F0/Fs;
    vsip_scalar_f w1 = 2 * M_PI * F1/Fs;
    vsip_scalar_f w2 = 2 * M_PI * F2/Fs;
    vsip_scalar_f w3 = 2 * M_PI * F3/Fs;
    vsip_scalar_f cnst1 = M_PI/Nnoise;
    vsip_offset offset0 = (vsip_offset)(alpha * Mp + 1);
    vsip_fftm_f *rcfftmop_obj = /* time fft */
    vsip_rcfftmop_create_f(Mp,Ns,1,VSIP_ROW,0,0);
    vsip_fftm_f *ccfftmip_obj = /*space fft */
          vsip_ccfftmip_create_f(Mp,Ns/2 +
              1,VSIP_FFT_FWD,1,VSIP_COL,0,0);
    vsip_vramp_f(0,1,m);
    vsip_vramp_f(0,M_PI/Mp,Xi);
    vsip_vcos_f(Xi,Xi);
    vsip_vouter_f(alpha,m,Xi,Xim);
    { vsip_vview_f *gram_v = vsip_mrowview_f(gram,0);
      vsip_vputlength_f(gram_v,Mp*(Ns/2 + 1));
      vsip_vfill_f(0,gram_v);
      vsip_vdestroy_f(gram_v);
    }
    for(j=0; j<Nnoise; j++){
       noise[j] = vsip_vcreate_f(Nn,0);
       vsip_vrandn_f(state,nv);
       vsip_firflt_f(fir,nv,noise[j]);
       vsip_svmul_f(12.0/(Nnoise),noise[j],noise[j]);
       vsip_vputlength_f(noise[j],Ns);
    }
    vsip_vramp_f(0,1.0,t); /* time vector */
```

Example 21 (3 of 4)

```
for(i=0; i<Mp; i++){
   vsip_scalar_f Xim_val = vsip_mget_f(Xim,i,Theta_o);
   data_v = vsip_mrowview_f(data,i);
   vsip_vsmsa_f(t,w0,-w0 * Xim_val,tt);
   vsip_vcos_f(tt,data_v); /*F0 time series */
   vsip_vsmsa_f(t,w1,-w1 * Xim_val,tt);
   vsip_vcos_f(tt,tt); /*F1 time series */
   vsip_vadd_f(tt,data_v,data_v);
   vsip_vsmsa_f(t,w2,-w2 * Xim_val,tt);
   vsip_vcos_f(tt,tt); /*F2 time series */
   vsip_vadd_f(tt,data_v,data_v);
   vsip_vsmsa_f(t,w3,-w3 * Xim_val,tt);
   vsip_vcos_f(tt,tt); /*F3 time series */
   vsip_svmul_f(3.0,tt,tt); /* scale by 3.0 */
   vsip_vadd_f(tt,data_v,data_v);
   vsip_svmul_f(3,data_v,data_v);
   for(j=0; j<Nnoise; j++){
       /* simple time delay beam forming for noise */
      vsip_vputoffset_f(noise[j],offset0 +
        (int)( i * alpha * cos(j * cnst1)));
      vsip_vadd_f(noise[j],data_v,data_v);
   }
   /* need to destroy before going on to next phone */
   vsip_vdestroy_f(data_v);
}
/* window the data and the array to reduce sidelobes */
vsip_vmmul_f(windowt,data,VSIP_ROW,data);
vsip_vmmul_f(windowp,data,VSIP_COL,data);

/* do ffts */
vsip_rcfftmop_f(rcfftmop_obj,data,gram_data);
vsip_ccfftmip_f(ccfftmip_obj,gram_data);
```

Example 21 (4 of 4)

```
   { /* scale gram to db, min 0 max 255 */
      vsip_index ind;
      gram_v = vsip_mrowview_f(gram,0);
      gram_data_v = vsip_cmcolview_f(gram_data,0);
      rmview = vsip_mrealview_f(gram_data);
      vsip_vputlength_f(gram_v,Mp*(Ns/2 + 1));
      vsip_cvputlength_f(gram_data_v,Mp*(Ns/2 + 1));
      data_v = vsip_vrealview_f(gram_data_v);
      vsip_vcmagsq_f(gram_data_v,data_v);
      vsip_mcopy_f_f(rmview,gram);
      vsip_vdestroy_f(data_v);
      vsip_svadd_f(1.0 - vsip_vminval_f(
                 gram_v,&ind),gram_v,gram_v);
      vsip_vlog10_f(gram_v,gram_v);
      vsip_svmul_f(256.0 / vsip_vmaxval_f(gram_v,&ind),
                    gram_v,gram_v);/* scale */
      /* reorganize the data to place zero in the
          center for direction space */
      data_v = vsip_vcloneview_f(gram_v);
      vsip_vputlength_f(data_v,(Mp/2) * (Ns/2 + 1));
      vsip_vputoffset_f(data_v,(Mp/2) * (Ns/2 + 1));
      vsip_vputlength_f(gram_v,(Mp/2) * (Ns/2 + 1));
      vsip_vswap_f(data_v,gram_v);
      vsip_vdestroy_f(gram_v);
      vsip_vdestroy_f(data_v);
      vsip_cvdestroy_f(gram_data_v);
      vsip_mdestroy_f(rmview);
   }
   VU_mprintgram_f(gram,"gram_output");
   return 0;
}
void VU_mprintgram_f(vsip_mview_f* M,char* fname)
{   vsip_length RL = vsip_mgetrowlength_f(M);
    vsip_length CL = vsip_mgetcollength_f(M);
    FILE *of = fopen(fname,"w");
    vsip_length row,col;
    for(row = 0; row<CL; row++)
       for(col=0; col<RL; col++)
         fprintf(of,"%ld %ld %3.0f\n",
             row,col,vsip_mget_f(M,row,col));
    fclose(of);
    return;
}
```

Figure 5 Coherent plane wave propagation in a 2D medium

$$\vec{k} = ((2\pi)/\lambda)(\cos(\theta)\hat{e}_x + \sin(\theta)\hat{e}_y)$$

$$c = (2\pi f)/\lambda = \omega/\lambda$$

$$p(t, \vec{r}) = p_o \exp(j(\vec{k} \cdot \vec{r}))\exp(-j2\pi f t)$$

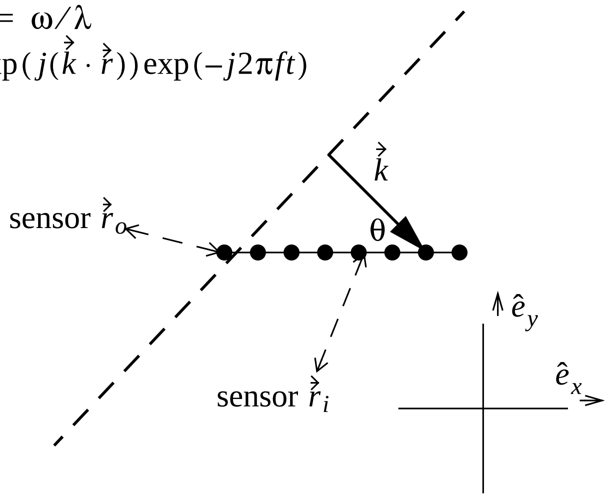
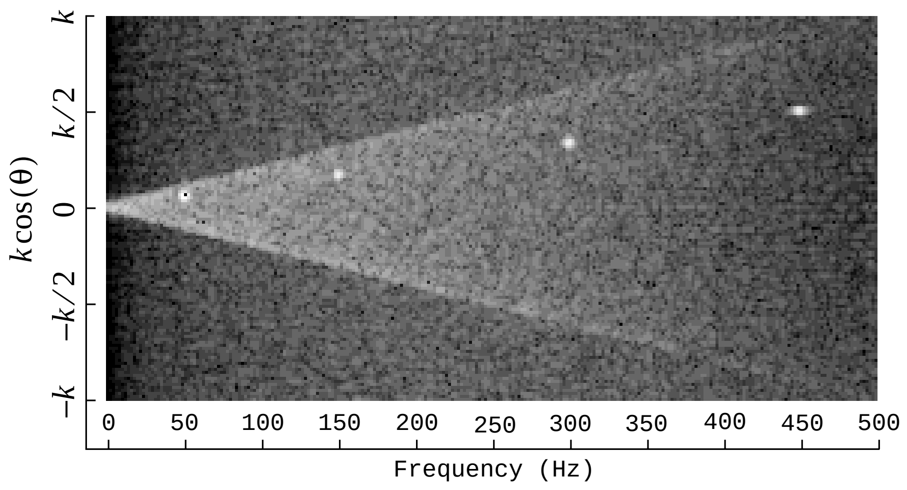


Figure 6 $k\omega$ for example 21



CHAPTER 7 Linear Algebra Functionality in the VSIPL Core Profile

Introduction

The VSIPL Core profile specifies support simple matrix operations such as matrix products, methods to solve the standard matrix equation $\bar{A}\vec{x} = \vec{b}$, and methods to solve least squares problems. VSIPL hides the decomposition of matrices in objects. So in addition to standard matrix products, special functions for doing matrix products with decomposition matrices are provided.

We note that although vectors are treated as column vectors in equations, VSIPL vector views have only one stride and so the action of the vector within the function is defined only by the function definition.

In general all matrix views passed into a function are defined as type **`const`**. This means that the area of the block mapped by the view does not change inside of the function call. For some of the defined in place operations where the input and output are defined by the same view the input matrix size may be different than that required by the output data. For these cases the strides of the input view define where the output data is placed. The first element of the output data replaces the first element of the input data. The author recommends defining a view of the output data space for convenience. For a couple of cases the output data space may be bigger than the input data space. Defining an output data view will ensure that the strides of the input view and the size of the block are sufficient to hold the output data.

Simple Matrix-Matrix and Vector-Matrix Operations

These matrix products include matrix products, vector matrix products, matrix vector products, outer products, vector dot products, a general matrix product, and a general matrix sum.

Most of the simple matrix products may not be done in place. An exception is **`vsip_cmherm_f`** which may be done in place only if the matrix is square.

The author notes that the dot product functionality of VSIPL has a complex dot product, **`vsip_cvdot_f`**, and a complex conjugate dot product, **`vsip_cvjdot_f`**. The user is cautioned to not become confused. The complex dot product is a simple complex multiply and add. The complex conjugate dot product is the more common complex multiply times complex conjugate and add.

Simple Solvers

VSIPL has simple one function calls to solve a covariance problem, a linear least square problem, or a Toeplitz symmetric positive definite symmetric problem. The decision whether to use a simple solver, or one of the more complicated solvers will depend on the application.

Covariance Problem

The function **`vsip_covsol_f`** solves a matrix equation of the form $\bar{A}^T\bar{A}\bar{X} = \bar{B}$ where A is of size M by N with $M \geq N$. We note that $\bar{A}^T\bar{A}$ is of size N by N so the input matrix $\bar{B}$ is of size N by N. This function is done in place and the output data replaces the input data. For this function the output data will exactly fit in the input view.

It is possible that there will be a memory allocation failure with this function. If this happens the function returns a value of negative one. If successful it returns a value of zero. A positive return value indicates a failure for some other reason.

Linear Least Squares Problem

The function **`vsip_llsqsol_f`** solves a matrix equation of the form $\bar{A}\bar{X} = \bar{B}$ where $\bar{A}$ is of size M by N and $M \geq N$. Generally this problem is overdetermined and $\bar{X}$ is solved for in the linear least squares sense. The least squares problem is well covered in many texts and the author will not go into the details.

The input matrix $\bar{B}$ is of size M by N and the output data is of size N by N. This means that the output data will reside in the top part of the input/output view. A view of the output data matrix may be created by using the matrix sub view function. We note that a view of the output data has the first element at index 0,0 of the input data, and has the same strides as the input data.

```
/* example of output view for vsip_llsqsol_f */
vsip_mview_f *output_data = vsip_msubview_f(
                                input_data, 0,0,N,N);
```

This function returns zero on success, negative one on failure due to a memory allocation problem, and positive for failure for some other reason.

Toeplitz System

This function, **`vsip_toepsol_f,`** solves a matrix equation of the form $\bar{A}\vec{x} = \vec{b}$ where the matrix $\bar{A}$ has the special form known as Toeplitz, where the diagonals are constant. In addition the matrix must be Hermitian and positive definite. This type matrix looks as follows for the Hermitian case. The real case is the obvious generalization. Note that r^* is the conjugate of r. Since a Hermitian matrix requires $\bar{A}^H = \bar{A}$ we see that the diagonal must have a zero imaginary component.

$$\begin{bmatrix} r_0 & r_1 & r_2 & \dots & r_{N-2} & r_{N-1} \\ r_1^* & r_0 & r_1 & \dots & \dots & r_{N-2} \\ r_2^* & r_1^* & r_0 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ r_{N-2}^* & \dots & \dots & \dots & r_0 & r_1 \\ r_{N-1}^* & r_{N-2}^* & \dots & \dots & r_1^* & r_o \end{bmatrix}$$

This functions only has vector arguments. Since all the required information for the matrix resides in the first row, this is the input argument. All the vectors input into this function may be overwritten in the output calculation. If the data is needed again it must be copied to a safe place before being input to the function. There is no in-place operation for this function.

The TASP VSIPL library uses the Levinson routine to solve this problem.

This function returns negative one if it fails because of a memory allocation problem, it returns zero if it completes successfully, and it returns a positive number if it fails for some other reason.

LU decomposition function set

The general matrix equation $\bar{A}\bar{X} = \bar{B}$ where $\bar{A}$ is a square matrix is solved with the LU (lower, upper) decomposition. This method is well covered in many texts and the author will not go into details. In VSIPL the solution follows a four step procedure.

First an object, called an LUD object, is created. This object is opaque and vendor dependent. It is designed to hold the decomposition, and any information or data space required when solving the matrix equation. If an allocation failure occurs when creating the object then the create function returns null.

Second the LUD object and the matrix to be decomposed are passed into the decomposition function. The matrix view is **`const`**, however the data space used by the matrix may be used by the decomposition. After the decomposition the matrix data space is associated with the LUD object and should not be used for any purpose. Note that this is only the data space associated with the matrix view. If the matrix is bound to a block with more view bound to it than just the matrix, portions of the block not mapped by the matrix view are not affected. The matrix continues to reside in memory, and the view continues to be available. They just should not be used until the LUD object is destroyed, at which time they may be reused or destroyed as required. The original data is of course not in the view, even after the LUD object is destroyed.

Third the matrix equation is solved. Note that the vsip_lusol_f function has a flag, the vsip_mat_op flag, that allows solving $\bar{A}\bar{X} = \bar{B}$ or $\bar{A}^T\bar{X} = \bar{B}$ for the real case, and $\bar{A}\bar{X} = \bar{B}$ or $\bar{A}^H\bar{X} = \bar{B}$ for the complex case. The TASP VSIPL library will also solve the matrix conjugate case, but this is no longer compliant with the current VSIPL library specification and should not be used. It will be removed from the library when the author gets around to rewriting the LUD function set.

It should be noted that LUD was written long ago without much research. It is not one of the more efficient or better written functions in the TASP VSIPL implementation. However it does appear to give the correct answer.

The fourth step is to destroy the LUD object after it is no longer needed.

Example 22 solves a simple problem using the LUD function set.

Example 22

```
/* A simple LUD example */
#include<vsip.h>
int main()
{
   vsip_mview_f *A =
        vsip_mcreate_f(3,3,VSIP_ROW,VSIP_MEM_NONE),
        *B = vsip_mcreate_f(3,2,VSIP_COL,VSIP_MEM_NONE);
   vsip_lu_f *lud = vsip_lud_create_f(3);
   vsip_mput_f(A,0,0,1);vsip_mput_f(A,0,1,2);
   vsip_mput_f(A,0,2,3);
   vsip_mput_f(A,1,0,-1);vsip_mput_f(A,1,1,1);
   vsip_mput_f(A,1,2,-2);
   vsip_mput_f(A,2,0,1);vsip_mput_f(A,2,1,1);
   vsip_mput_f(A,2,2,-3);
   vsip_mput_f(B,0,0,1); vsip_mput_f(B,0,1,1);
   vsip_mput_f(B,1,0,-1); vsip_mput_f(B,1,1,1);
   vsip_mput_f(B,2,0,-2); vsip_mput_f(B,2,1,3);
   printf(“A = “); VU_mprintm_f(“%4.2f ”,A);
   printf(“B = “); VU_mprintm_f(“%4.2f ”,B);
   if(0 != vsip_lud_f(lud,A)) return 1;
   vsip_lusol_f(lud,VSIP_MAT_NTRANS,B);
   printf(“X = “); VU_mprintm_f(“%6.3f ”,B);
   vsip_malldestroy_f(A); vsip_malldestroy_f(B);
   vsip_lud_destroy_f(lud);
   return 0;
}
```

The output for example 22 (with liberal formatting) looks as follows:

```
A = [1.00 2.00 3.00
      -1.00 1.00 -2.00
      1.00 1.00 -3.00 ];

B = [1.00 1.00
      -1.00 1.00
     -2.00 3.00 ];

X = [-0.235  0.765
       -0.176  0.824
       0.529 -0.471 ];
```

Cholesky Decomposition Function set

The author did not write the Cholesky Decomposition currently in the library, and is not familiar with the properties Cholesky. The function set does appear to work, and is compliant with the specification. The author will become more familiar with Cholesky in time, but for now is not prepared to write this section, and so will blow it of until some future release.

QR Decomposition Function set

The QR decomposition function set is similar to the LU decomposition; however it is much more complicated with extra functions. The LU decomposition solves the fully determined matrix equation $\bar{A}\bar{X} = \bar{B}$ where $\bar{A}$ is square. The QR decomposition is used to solve the overdetermined matrix equation where $\bar{A}$ is of size M by N where $M \geq N$. There are also methods to solve equation with the $\bar{R}$ matrix from the decomposition, ignoring the $\bar{Q}$ matrix altogether, and a method to do matrix products with the $\bar{Q}$ matrix. The author is not expert (or even slightly competent) on the uses of QR decomposition, and so will not go into many details on its use here.

For QRD the author has used only the Householder method. There are options to only solve for skinny $\bar{Q}$, or for the full $\bar{Q}$, or for just $\bar{R}$. However for the current TASP VSIPL implementation it is always done with Householder's method. The function may fail if the proper options are not selected, as they are tested for. For instance if $\bar{Q}$ is not required to be saved in the decomposition, calling a function which requires $\bar{Q}$ will fail. Householder, used properly, seems to cover all the bases, although it may not be the most efficient or best way for every case.

The first two steps in using the QR decomposition function set are the same as the LUD function set.

The first step is to create the decomposition object. During the creation of the object clues are passed in as to what the user requires from the decomposition. These options require either just the $\bar{R}$ matrix, or the $\bar{R}$ matrix and either the skinny $\bar{Q}$ (also called $\overline{Q1}$) matrix or the full $\bar{Q}$ matrix. For explanations of skinny and full matrix please refer to a linear algebra text.

The second step is the decomposition. The decomposition function takes the QRD object, and the input matrix A and does the decomposition. The same as LUD the $\bar{A}$ data matrix (data space) is owned by the QRD object and should not be touched after the decomposition. The contents of A is not defined by the specification, and should not be used directly. Even if the contents of $\bar{A}$ are recognizable the contents are vendor dependent, and any code produced which uses the contents, will not be portable. The QRD object must be used along with the QRD function set.

Once the decomposition has taken place three functions may be called. These are a QRD product function, a linear system solver based on $\bar{R}$, and a covariance or linear least squares solver based.

For example 23 we use the linear system solver to solve the LUD example22.

Example 23

```
/* A simple QRD example */
#include<vsip.h>
int main()
{
   vsip_mview_f *A =
        vsip_mcreate_f(3,3,VSIP_ROW,VSIP_MEM_NONE),
        *B = vsip_mcreate_f(3,2,VSIP_COL,VSIP_MEM_NONE);
   vsip_qr_f *qrd = vsip_qrd_create_f(3,3,VSIP_QRD_SAVEQ);
   vsip_mput_f(A,0,0,1);vsip_mput_f(A,0,1,2);
   vsip_mput_f(A,0,2,3);
   vsip_mput_f(A,1,0,-1);vsip_mput_f(A,1,1,1);
   vsip_mput_f(A,1,2,-2);
   vsip_mput_f(A,2,0,1);vsip_mput_f(A,2,1,1);
   vsip_mput_f(A,2,2,-3);
   vsip_mput_f(B,0,0,1); vsip_mput_f(B,0,1,1);
   vsip_mput_f(B,1,0,-1); vsip_mput_f(B,1,1,1);
   vsip_mput_f(B,2,0,-2); vsip_mput_f(B,2,1,3);
   printf("A = "); VU_mprintm_f("%4.2f ",A);
   printf("B = "); VU_mprintm_f("%4.2f ",B);
   if(0 != vsip_qrd_f(qrd,A)) return 1;
   vsip_qrsol_f(qrd,VSIP_LLS,B);
   printf("X = "); VU_mprintm_f("%6.3f ",B);
   vsip_malldestroy_f(A); vsip_malldestroy_f(B);
   vsip_qrd_destroy_f(qrd);
   return 0;
}
```

We get the same answer as example 22 (hopefully). We note that in **line 8** for example 22 only a single number is passed, but in example 23 we have two numbers and an option. The option says to return an LUD object suitable for calculating a full Q. The LUD input matrix is square so only the size of the matrix is needed, but the QRD input matrix may have more rows than columns so the first number is the column length, and the second number is the row length. Since the QRD solver solves both the least squares problem, and the covariance problem we need to tell it which one to solve. This is what the flag **`VSIP_LLS`** does in **line 21.**

The Q product function

The function **`vsip_qrdprodq_f`** supports the product of a matrix with the $\overline{Q}$ from the QRD decomposition. Although we don't have the actual $\overline{Q}$, just the QRD object, the $\overline{Q}$ still has an understood matrix size and the size of the input matrix will depend on it for the matrix product to be conformant. The size of the $\overline{Q}$ will be M by N (the size of the decomposed matrix) if the skinny Q save option is selected. If the full Q save option is selected then $\overline{Q}$ will be of size M by M. The Q product function allows $\overline{Q}$ or $\overline{Q}^T$ ($\overline{Q}^H$)to multiply the input matrix on the right, or to be multiplied by the input matrix on the left. So the following cases are possible.

Real Cases for $\overline{Q}$ product				
Options	Operation	Left Side	Right Side	Output Size
VSIP_QRD_SAVEQ **VSIP_NTRANS** **VSIP_MAT_LSIDE**	$\overline{Q}\overline{B}$	M by M	M by K	M by K
VSIP_QRD_SAVEQ **VSIP_TRANS** **VSIP_MAT_LSIDE**	$\overline{Q}^T\overline{B}$	M by M	M by K	M by K
VSIP_QRD_SAVEQ **VSIP_NTRANS** **VSIP_MAT_RSIDE**	$\overline{B}\overline{Q}$	K by M	M by M	K by M
VSIP_QRD_SAVEQ **VSIP_TRANS** **VSIP_MAT_RSIDE**	$\overline{B}\overline{Q}^T$	K by M	M by M	K by M
VSIP_QRD_SAVEQ1 **VSIP_NTRANS** **VSIP_MAT_LSIDE**	$\overline{Q}\overline{B}$	M by N	N by K	M by K (*Bigger*)
VSIP_QRD_SAVEQ1 **VSIP_TRANS** **VSIP_MAT_LSIDE**	$\overline{Q}^T\overline{B}$	N by M	M by K	N by K

Real Cases for $\bar{Q}$ product				
Options	Operation	Left Side	Right Side	Output Size
VSIP_QRD_SAVEQ1 **VSIP_NTRANS** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}$	K by M	M by N	K by N
VSIP_QRD_SAVEQ1 **VSIP_TRANS** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}^T$	K by N	N by M	K by M (*Bigger*)

Complex Cases for $\bar{Q}$ product				
Options	Operation	Left Side	Right Side	Output Size
VSIP_QRD_SAVEQ **VSIP_NTRANS** **VSIP_MAT_LSIDE**	$\bar{Q}\bar{B}$	M by M	M by K	M by K
VSIP_QRD_SAVEQ **VSIP_HERM** **VSIP_MAT_LSIDE**	$\bar{Q}^H\bar{B}$	M by M	M by K	M by K
VSIP_QRD_SAVEQ **VSIP_NTRANS** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}$	K by M	M by M	K by M
VSIP_QRD_SAVEQ **VSIP_HERM** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}^H$	K by M	M by M	K by M
VSIP_QRD_SAVEQ1 **VSIP_NTRANS** **VSIP_MAT_LSIDE**	$\bar{Q}\bar{B}$	M by N	N by K	M by K (*Bigger*)
VSIP_QRD_SAVEQ1 **VSIP_HERM** **VSIP_MAT_LSIDE**	$\bar{Q}^H\bar{B}$	N by M	M by K	N by K
VSIP_QRD_SAVEQ1 **VSIP_HERM** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}$	K by M	M by N	K by N
VSIP_QRD_SAVEQ1 **VSIP_HERM** **VSIP_MAT_RSIDE**	$\bar{B}\bar{Q}^H$	K by N	N by M	K by M (*Bigger*)

We note that two cases have been marked *Bigger.* This means that the output data will not fit in the input view. For these cases the user must be careful that the strides of the input matrix view will also work for a view of the output matrix data. If the input matrix is created as a sub view of the output matrix this will take care of any problems. For example for the case of save skinny Q, no transform on the left we have the following:

```
/* create an input view with sufficient stride sizes */
vsip_mview_f *output_view =
                vsip_mcreate_f(M,K,VSIP_ROW,VSIP_MEM_NONE);
vsip_mview_f *input_view =
               vsip_msubview_f(output_view,0,0,N,K);
```

For example 24 we demonstrate the Q product by using some properties of the QR decomposition. We know that $\bar{Q}$ is orthonormal, that $\bar{A} = \bar{Q}\bar{R}$ and that $\bar{R} = \bar{Q}^T\bar{A}$. We use the identity matrix to extract an estimate of Q.

In **lines 12-17** we input our matrix, **lines 18-23** initializes the Q matrix to the identity. We do the decomposition in **line 25**, and in **line 16** we multiply the identity matrix in the Q matrix by $\bar{Q}$ in place. We had already created a transpose view of the Q matrix in **line 8**. In **line 28** we use a matrix product function to multiply the transpose of $\bar{Q}$ times $\bar{A}$ giving $\bar{R}$. We then use the QRD product function in **line 31** to see if we can get back the original A by multiplying $\bar{Q}$ and $\bar{R}$ together.

The output for example 24 (with some formatting) follows:

```
A = [1.00 2.00 3.00
    -1.00 1.00 -2.00
    1.00 1.00 -3.00 ];

Q = I Q =[ 0.577  0.617  0.535
          -0.577  0.772 -0.267
          0.577  0.154 -0.802 ];

R = QT A =[ 1.732  1.155  1.155
           -0.000  2.160 -0.154
          -0.000 -0.000  4.543 ];

A = QR =[ 1.000  2.000  3.000
          -1.000  1.000 -2.000
          1.000  1.000 -3.000 ];
```

Example 24

```
/* A simple Q product example */
#include<vsip.h>
int main()
{
  vsip_mview_f
      *A  = vsip_mcreate_f(3,3,VSIP_ROW,VSIP_MEM_NONE),
      *Q  = vsip_mcreate_f(3,3,VSIP_ROW,VSIP_MEM_NONE),
      *QT = vsip_mtransview_f(Q),
      *A0 = vsip_mcreate_f(3,3,VSIP_COL,VSIP_MEM_NONE),
      *R  = vsip_mcreate_f(3,3,VSIP_COL,VSIP_MEM_NONE);
  vsip_qr_f *qrd=vsip_qrd_create_f(3,3,VSIP_QRD_SAVEQ1);
  vsip_mput_f(A,0,0,1);vsip_mput_f(A,0,1,2);
  vsip_mput_f(A,0,2,3);
  vsip_mput_f(A,1,0,-1);vsip_mput_f(A,1,1,1);
  vsip_mput_f(A,1,2,-2);
  vsip_mput_f(A,2,0,1);vsip_mput_f(A,2,1,1);
  vsip_mput_f(A,2,2,-3);
  vsip_mput_f(Q,0,0,1);vsip_mput_f(Q,0,1,0);
  vsip_mput_f(Q,0,2,0);
  vsip_mput_f(Q,1,0,0);vsip_mput_f(Q,1,1,1);
  vsip_mput_f(Q,0,2,0);
  vsip_mput_f(Q,2,0,0);vsip_mput_f(Q,2,1,0);
  vsip_mput_f(Q,2,2,1); vsip_mcopy_f_f(A,A0);
  printf("A = "); VU_mprintm_f("%4.2f ",A);
  if(0 != vsip_qrd_f(qrd,A)) return 1;
  vsip_qrdprodq_f(qrd,VSIP_MAT_NTRANS,VSIP_MAT_RSIDE,Q);
  printf("Q = I Q =");VU_mprintm_f("%6.3f ",Q);
  vsip_mprod_f(QT,A0,R);
  printf("R = QT A =");VU_mprintm_f("%6.3f ",R);
  vsip_qrdprodq_f(qrd,VSIP_MAT_NTRANS,VSIP_MAT_LSIDE,R);
  printf("A = QR =");VU_mprintm_f("%6.3f ",R);
  vsip_malldestroy_f(A); vsip_malldestroy_f(A0);
  vsip_malldestroy_f(R);
  vsip_mdestroy_f(QT); vsip_malldestroy_f(Q);
  vsip_qrd_destroy_f(qrd);
  return 0;
}
```

The R solver function

The QRD solve R function, `vsip_qrdsolr_f`, solves a linear system of the form $\bar{R}\bar{X} = \alpha\bar{B}$ or $\bar{R}^T\bar{X} = \alpha\bar{B}$ where $\bar{R}$ is the upper triangular matrix from the QR decomposition and alpha is a scalar. The calculation is done in place so that input matrix $\bar{B}$ is replaced by output matrix $\bar{X}$. Since $\bar{R}$ is square the input and output are exactly the same size.

Final Remarks for Linear Algebra.

The decomposition functions in linear algebra are not well tested. The author does not guarantee, promise, or think the routines are stable for ill conditioned matrices. The routines also may not be very efficient. The author intends to keep working on the algorithms with the goal of reasonable efficiency, and good numerical properties. For the present (and even the future) the user is cautioned to be suspicious of any results. The author, as always, welcomes feedback and advice on these functions, or any functions in the TASP VSIPL library.

INDEX

Appendix A VSIPL Fundamentals

VSIPL Fundamentals

Introduction

This appendix contains fundamental information about a VSIPL compliant library including the basic type definitions, *block* requirements, *view* requirements, and basic VSIPL definitions and terminology.

Note that there are various requirements in the functionality document which pertain to some small subset of functions. These requirements are not covered here. This document covers the more general requirements that must be met by virtually all VSIPL implementations, no matter the *profile*.

Disclaimer

The VSIPL library is *object based*, **not** *object oriented*. The reader should be careful to not bring along to this document any object oriented terms which may have specific meaning to them from another context, but which are being used by the VSIPL forum to mean something else. The basic terminology used by VSIPL is described below.

Blocks and Views

VSIPL has a notion of data storage in a *block*. A *block* is an abstract notion of contiguous data elements available for storage of data. Associated with a block is a block object. The block object contains the information necessary for the VSIPL implementation to access and the memory needed by the block for data storage. The design of the block object is implementation dependent.

VSIPL has a notion of vectors, matrices and three tensors which are *views* of the block. The information necessary to access the block data as if it is a vector, matrix or tensor is stored in the view object.

User Data Arrays, VSIPL Data Arrays, Released and Admitted

Memory allocated by VSIPL for data storage is termed a VSIPL data array. There is no method for a user to directly access a VSIPL data array. This causes a problem when input or output of data is needed from VSIPL. To address this problem functions are available in VSIPL to associate a data array allocated by the user to a VSIPL block.

To insure that use of data stored in a user data array by the application does not conflict with use of the data by VSIPL functions the block object associated with the data array maintains state information which informs VSIPL whether or not the user block is *admitted* or *released*.

Note a *user block* is a block which is associated with a user data array. A *VSIPL block* is a block which is not associated with a user data array. These are states of a block. The block type of a user block and a VSIPL block are identical.

Functions are available to admit or release a user block. When a block is *admitted* it is an error to directly manipulate any data in the user data array. When a block is *released* it is an error to use any VSIPL function which will read or write data in the block.

VSIPL Naming Convention and Functionality Requirements

While there is nothing to prevent an implementor from writing VSIPL compatible functions, only those functions that have been approved and are included in formal VSIPL documentation are a part of VSIPL. Functions outside of the standard should not use the VSIPL naming conventions to avoid confusion of porting of applications. In particular, function names outside of VSIPL should not start with "vsip_".

All VSIPL functionality is called out explicitly in the functionality document except for precision. The need to allow wide variation in precision to support diverse hardware precluded any attempt to specify every possible data precision. Specified methods must be followed for including data precision using a precision affix at the end of the specified VSIPL name. The approved affixes are covered in the summary of VSIPL types below. Except for copies the precision affix is usually a suffix. Copies require two precision affix's to make up the precision suffix.

Summary of VSIPL Types

All VSIPL type declarations and function names have the data type encoded into the name. The following are required VSIPL affix notations for use in encoding type data in the names and type declarations, and a description of the data types supported in VSIPL. It is not expected that any implementation will support all possible VSIPL data types. The data type supported will depend in part on the hardware for which the library was developed, and the expected use of the hardware.

Note that throughout the VSIPL documentation an affix of *_p* is used to denote a general precision of any type, and is a method to name functions or data types without having to spell out every single prefix which might be needed for that function or data type. Also used are *_i* to denote any integer, or an *_f* to denote any float. Note that the generalized affix is in a different font style than the specific affix. To produce a valid VSIPL name, or data type, use a specified name, or data type, from the functionality document, and replace the generalized affix with the selected affix from the table below.

Standard Floating Point Data Types	
Affix	**Definition**
`_f`	ANSI C single precision floating point
`_d`	ANSI C double precision floating point
`_l`	ANSI C extra precision floating point

Standard Integer Data Types	
Affix	**Definition**
`_c`	ANSI C char
`_uc`	ANSI C unsigned char
`_si`	ANSI C short integer

Standard Integer Data Types	
Affix	**Definition**
`_us`	ANSI C unsigned short integer
`_i`	ANSI C integer
`_u`	ANSI C unsigned integer
`_li`	ANSI C long integer
`_ul`	ANSI C unsigned long integer
`_ll`	Long, long integer, implementation dependent
`_ull`	Unsigned long, long integer, implementation dependent

Portable Precision Floating Point Data Types	
Affix	**Definition**
`_f6`	Floating point type with at least 6 decimal digits of accuracy. IEE 754 single precision (32 bit) has 6 decimal digits of accuracy.
`_f15`	Floating point types with at least 15 decimal digits of accuracy. IEEE 754 double precision (64 bit) has 15 decimal digits of accuracy.
`_f`*n*	Floating point type with at least *n* decimal digits of accuracy. If the system supports such a precision, it resolves to the smallest C type based on the values of `FLT_MANT_DIG`, `DBL_MANT_DIG`, or `LDBL_MANT_DIG` (which are defined in `float.h`).

Portable Precision Integer Data Types	
Affix	**Definition**
`_il8`	*int* of at least 8 bits
`_il16`	*int* of at least 16 bits
`_il32`	*int* of at least 32 bits
`_il64`	*int* of at least 64 bits
`_il`*n*	*int* of at least *n* bits
`_ul8`	unsigned *int* of at least 8 bits
`_ul16`	unsigned *int* of at least 16 bits
`_ul32`	unsigned *int* of at least 32 bits
`_ul64`	unsigned *int* of at least 64 bits
`_ul`*n*	unsigned *int* of at least *n* bits

Portable Precision Integer Data Types

Affix	Definition
`_ie8`	*int* of exactly 8 bits
`_ie16`	*int* of exactly 16 bits
`_ie32`	*int* of exactly 32 bits
`_ie64`	*int* of exactly 64 bits
`_ie`*n*	*int* of exactly *n* bits
`_ue8`	unsigned *int* of exactly 8 bits
`_ue16`	unsigned *int* of exactly 16 bits
`_ue32`	unsigned *int* of exactly 32 bits
`_ue64`	unsigned *int* of exactly 64 bits
`_ue`*n*	unsigned *int* of exactly *n* bits
`_if8`	fastest *int* of at least 8 bits
`_if16`	fastest *int* of at least 16 bits
`_if32`	fastest *int* of at least 32 bits
`-if64`	fastest *int* of at least 64 bits
`_if`*n*	fastest *int* of at least *n* bits
`_uf8`	unsigned fastest *int* of at least 8 bits
`_uf16`	unsigned fastest *int* of at least 16 bits
`_uf32`	unsigned fastest *int* of at least 32 bits
`_uf64`	unsigned fastest *int* of at least 64 bits
`_uf`*n*	unsigned fastest *int* of at least *n* bits

Other Data Types

Affix	Definition
_bl	Boolean Data Type. Logical *false* for 0, and Logical *true* for non-zero.
_vi	Vector Index. This is an unsigned integer of sufficient precision to index any VSIPL vector.
_mi	Matrix Index. This is a data type used for accessing matrix elements. The precision of the type is the same as _vi. The *matrix index* of the element $x_{i,j}$ is the 2-tuple $\{i, j\}$.
_ti	Tensor Index. This is a data type used for accessing tensor elements. The precision of the type is the same as _vi. The *tensor index* of the element $x_{i,j,k}$ is the 3-tuple $\{i, j, k\}$.

Basic Data Types

VSIPL has three basic data types, *scalars*, *blocks*, and *views*. VSIPL also has other special data types and structures, used for defining special objects, which are used in a single function or a small subset of functions. These special structures and data types are defined in the functionality document, but not here. Structures required for VSIPL block creation are defined below. Also defined below are structures for complex scalars and scalar indices.

Scalar Data Types

All supported VSIPL scalars have a type definition of
`vsip_scalar_`*p*
for real scalars, and a type definition of
`vsip_cscalar_`*p*
for complex scalars. Complex scalars are only supported for float and integer data types.

Note: For VSIPL 1.0 there are support functions for complex integers, but no other functions which use complex integers are defined.

Below find an example of a VSIPL header definition for a scalar float, and a scalar unsigned integer.

```
typedef float vsip_scalar_f;
typedef unsigned int vsip_scalar_u;
```

The following definitions (if the type is supported) must be included with the library. Some of the information is implementation dependent. Implementation dependent information will be indicated with a bracket (<?...?>) around the dependent section

Complex	`typedef struct {vsip_scalar_`*p* `r, i;} vsip_cscalar_`*p*`;`

Boolean	`typedef <?char?> vsip_scalar_bl;`
	`typedef vsip_scalar_bl vsip_bool;`
	`#define VSIP_FALSE 0`
	`#define VSIP_TRUE 1`

Vector index	`typedef unsigned <?long int?> vsip_scalar_vi`
	`typedef vsip_scalar_vi vsip_index;`

Matrix index	`typedef struct {vsip_scalar_vi r,c;} vsip_scalar_mi;`
Tensor index	`typedef struct {vsip_scalar_vi l,r,c;} vsip_scalar_ti;`
Offset	`typedef vsip_scalar_vi vsip_offset;`
Stride	`typedef signed <?long int?> vsip_stride;`
length	`typedef vsip_scalar_vi vsip_length;`

Note: The data type for the vector index (`vsip_scalar_vi`) is implementation dependent. It must be an *unsigned* integer of sufficient size to allow indexing any possible view element of the implementation.

Note: The stride data type must be a *signed* integer of the same number of bits precision as the vector index.

Block Data Types

All supported VSIPL blocks have a type definition as described in the following table.

Type	**VSIPL blocks**
`vsip_block_`*p*	For real blocks, index blocks, and boolean blocks
`vsip_cblock_`*p*	For complex blocks. Complex blocks are only supported for float and integer data.

Examples of incomplete type definitions for blocks included in vsip.h

`struct vsip_blockobject_bl; /* boolean block structure */`
`typedef struct vsip_blockobject_bl` **`vsip_block_bl`**`;`

`struct vsip_blockobject_vi; /* vector index block structure */`
`typedef struct vsip_blockobject_vi` **`vsip_block_vi`**`;`

`struct vsip_blockobject_d; /* double block structure */`
`typedef struct vsip_blockobject_d` **`vsip_block_d`**`;`

`struct vsip_cblockobject_d; /* complex double block structure */`
`typedef struct vsip_cblockobject_d` **`vsip_cblock_d`**`;`

Note that the above structures `vsip_blockobject_bl`, `vsip_blockobject_vi`, `vsip_blockobject_d`, and `vsip_cblockobject_d` all may reside in a VSIPL header file which is private to the implementation developer. The names of all these structures are implementation dependent. The only required naming convention is the block type necessary for declaring VSIPL objects in the user application. For the examples above these are in bold.

The following hint structure must be included with an implementation. It is not required that the hints be supported (in the functions where they are required), but the structure must be available for portability reasons. Additional details of the hint are available on the functionality page where it is defined.

```
typedef enum {
                VSIP_MEM_NONE = 0,
                VSIP_MEM_RDONLY = 1,
                VSIP_MEM_CONST = 2,
                VSIP_MEM_SHARED = 3,
                VSIP_MEM_SHARED_RDONLY = 4,
                VSIP_MEM_SHARED_CONST = 5
                } vsip_memory_hint;
```

View Data Types

All supported VSIPL views have a type definition as follows

`vsip_vview_`*p*	For real vector views, boolean vector views and index vector views.
`vsip_cvview_`*p*	For complex vector views.
`vsip_mview_`*p*	For real matrix views and boolean matrix views.
`vsip_cmview_`*p*	For complex matrix views
`vsip_tview_`*p*	For real tensor views
`vsip_ctview_`*p*	For complex tensor views

Note that all index views are vectors. There are only types `vsip_vview_vi`, `vsip_vview_mi`, and types `vsip_vview_ti`.

Examples of incomplete type definitions for views

`struct vsip_vviewobject_bl; /* boolean vector view struct */`
`typedef struct vsip_vviewobject_bl` **`vsip_vview_bl`**`;`

`struct vsip_vviewobject_vi; /* vector index view struct */`
`typedef struct vsip_vviewobject_vi` **`vsip_block_vi`**`;`

`struct vsip_vviewobject_d; /* double vector view struct */`
`typedef struct vsip_blockobject_d` **`vsip_block_d`**`;`

`struct vsip_cvviewobject_d; /* complex double vector view struct */`
`typedef struct vsip_cvviewobject_d` **`vsip_cblock_d`**`;`

Note that the above structures `vsip_vviewobject_bl`, `vsip_vviewobject_vi`, `vsip_vviewobject_d`, and `vsip_cvviewobject_d` may all reside in a VSIPL header file which is private to the implementation developer. The names of all these structures are implementation dependent. The only required naming convention is the view type needed by the user to declare view objects. For the examples above these are in bold.

Block Requirements

A *block* is a VSIPL type representing an object where data is stored. The *block* is conceptually a one dimensional data array of elements of a single data type. Mixed data types are not supported. The user supplies the size of the block on its creation.

The data in a block is accessed using *views*. The attributes stored in a view describe a portion of a blocks data using offset from the beginning of the block, stride through the block, and number of elements of the block described by the view (the length attribute). The block location of the first element is at zero, and the block location of the final element is at N-1, where N is the total length of the block.

There are two kinds of blocks, *user* and *VSIPL*. A user block is one which is created using a data array which is allocated directly by the application so that the application has a pointer to the data array. A VSIPL block is one which is not associated with a user data array, and the user has no (proper) method to retrieve a pointer to the blocks data array.

A *user* block is either in a released or an admitted state to VSIPL. It is an error for a released *user* blocks data array to be accessed by any VSIPL function which will read or write elements in the data array. Access to a released *user* blocks data must be through direct manipulation of the data array. Access to an *admitted* user block must be through VSIPL functions. It is an error to directly manipulate or read a user blocks data array after it has been admitted to VSIPL. After admission only VSIPL functions should be used to access the data. A *VSIPL* block is created in the admitted state and can not be released. A *user* block is created in the released state, and can be admitted or released as required by the application.

A *VSIPL* block is created with a VSIPL creation function. When a VSIPL block is created, the data array bound to it is created at the same time. The details of the physical storage of the data array is implementation dependent; however the data array appears as contiguous data elements for the purpose of assigning strides and offsets in views of the block. *VSIPL* blocks are always admitted and the data array can only be manipulated with VSIPL function calls.

Except for a *released user block*, there is no function available to make a *block* and then at some later time attach a data array to it. It is possible to create a *user block* using a `NULL` data pointer. A *user block* bound to a `NULL` pointer can not be *admitted* until the *block* is re-bound to a data pointer which is not `NULL`. A *user block* in the *released* state can be re-bound to any valid pointer of the proper data type for the *block*. There is no function available in VSIPL 1.0 to allow rebinding an admitted block to different data array.

For *released user blocks* the *user data* must be contiguous with the following exception. For *complex user blocks* the attached data can be contiguous, or it can consist of two separate contiguous *data arrays* of equal size.

When a *released* block is admitted the implementation is free to do whatever is necessary without concern for the *released* data layout. The *user array data* layout must be restored to the same location and layout when the block is *released*.

A *user* block, *admitted* or *released*, and a *VSIPL* block, which contain data of the same type, have a single block data type. Any information needed by the implementation developer regarding the state of the block (admitted, released, user, VSIPL, etc.) is hidden from the application using some implementation dependent method such as *hidden attributes* of the block object.

Derived Blocks

There are functions defined to *derive real views* from *complex views*. These functions produce a *derived block* to bind the real views to. The derived block is of type `vsip_block_`*p*.

The *required derived block* data space must encompass the entire real portion of the complex block if a real view is derived, or the entire imaginary portion of the underlying complex block if an imaginary view is derived. The *derived block* can encompass other portions of the complex block outside the range of the required real or imaginary data space; however the implementation is only required to maintain views bound to the *required derived block* data space. It is an error to bind new views to a *derived block* which will encompass both real and imaginary portions of the original complex block. The result is implementation dependent.

The *derived* block is destroyed when the complex block is destroyed. It is an error to destroy a derived block directly. The implementation must maintain sufficient state information in the *complex* block, and the *derived* real block, to support the proper behavior of the *derived* block.

The internal format of any *admitted block* is implementation dependent so underlying memory layout of the complex block is unknown. The following conditions must be met by an implementation for derived blocks.

1. A *derived block* bound to a *user complex block* can not be *released*. The *derived block* is *released* when the *complex block* it is bound to is released.
2. A *derived block* bound to a *user complex block* can not be admitted. The *derived block* is *admitted* when the *complex block* it is bound to is *admitted*.

View Requirements

A *view* is a VSIPL type representing some portion of the data in a block. A block can have many *views* bound to it; however a *view* can only be bound to a single block.

When a *view* is created it is bound to a block There is no method in VSIPL 1.0 to create a *view* that is not bound to a *block*. A *view* must have a *user attribute* which defines the *block* the *view* is bound to. The *block attribute* is **not** setable.

A *view* must have an *offset attribute* which indicates the number of *elements* from the beginning of the *block* where the first *element* of the *view* is located within the block. The *offset attribute* is indexed starting at zero so that an offset of zero implies the first *element* of the view is the first *element* of the block. The offset attribute is setable by the application.

A view will have one or more stride attributes. The magnitude of the stride attribute defines the distance between two consecutive elements in some view dimension. For example, the row stride indicates how many elements (through the block) from one element in the row to the next element in the row. The distance defined is through the block. The sign of the stride attribute defines what direction the view description moves through the block as the index value of the view description increases. A stride of zero must be supported. The stride attribute is settable

A view will have a length attribute for each stride attribute. A length attribute is a positive integer describing the number of elements in the view direction, such as the number of elements in the row of a matrix. The length attribute is settable.

For vectors there is only one dimension so there is only one stride, and one length. For matrices there are two view dimensions, so there are two strides, and two lengths. For three tensors there are three strides and three lengths.

Additional information on offset, stride, and length attributes are available in the functionality document.

Complex Views and Derived Real Views.

It is required that it be possible to create a *derived view* of the real or imaginary portion of a complex *view*. Note that this is **not** a copy. Replacing an element in the *real* or *imaginary view* derived from the *complex view* replaces the corresponding element in the *complex view*. Similarly replacing a complex element in the *complex view* replaces the corresponding element in the *real* and *imaginary view*. The *real view* and *imaginary view* of the *complex view* are real and are not complex. They must be bound to a *block* of type `vsip_block_`*p*. The real block bound to the *real* or *imaginary view* of a *complex view* is termed a *derived block* since it is *derived* from the complex block.

The method of instantiating a *derived* block is implementation dependent.

Note: Because of the implementation dependent nature of *derived blocks* the *stride* and *offset* of *derived views* are **not** determined until after the *view* is created.

User Data

This section covers the required data layout of *user data arrays*. The implementation developer must support, and the application developer must use, the required data array formats for *user* data. These formats allow for portable input of user data into VSIPL, and portable output of VSIPL results to the application.

For **float** the user data array is a contiguous memory segment of type `vsip_scalar_`*f*.

For **integer** the user data array is a contiguous memory segment of type `vsip_scalar_`*i*.

For **boolean** the user data array is a contiguous memory segment of type `vsip_scalar_bl`.

For **vector index** the user data array is a contiguous memory segment of type `vsip_scalar_vi`.

For **matrix index** the user data array is a contiguous memory segment of type `vsip_scalar_vi`. The matrix index element in a user data array is two consecutive elements of type `vsip_scalar_vi`. The first element is the row index, the second is the column index. Note that the matrix index element in a user data array is not the same as `vsip_scalar_mi`. This corresponds to the *interleaved* method described below for complex.

For **tensor index** the user data array is a contiguous memory segment of type `vsip_scalar_vi`. The tensor index element in a user data array is three consecutive elements of type `vsip_scalar_vi`. The first value in the element is the leg index, the second is the row index, and the third is the column index. Note that the tensor index element in a user data array is not the same as `vsip_scalar_ti`.

For **complex float** or **complex integer** the user data array is either *interleaved* or *split* as described below. Both the interleaved and split formats must be supported for user data. Note that the data format for complex float *user data* arrays is not of type `vsip_cscalar_`*p*

> **Interleaved**: The user data array is a contiguous memory segment of type `vsip_scalar_`*p*. The complex element is two consecutive elements of type `vsip_scalar_`*p*. The first element is real, the second imaginary.
>
> **Split**: The user data array consists of two contiguous memory segments of equal length, each of type `vsip_scalar_`*p*. The order of the arguments when the data is bound to a block determines the real and imaginary portions.

Development mode requirements

The functionality portion of VSIP 1.0 has required error checks for *development* mode. The basic requirement is that the implementation developer of a library supporting development mode must maintain sufficient information within the implementation to support the required error checks for every function supported by the implementation.