JVSIP User Manual

Randall Judd

June 2, 2014

Draft

Preface

This book describes the functionality of the JVSIP implementation of the Vector/Signal/Image processing (VSIP) Library (VSIPL). JVSIP includes all the functionality of the TASP Core Plus implementation (TVCPP) developed by the author as part of the the TASP (Tactical Advanced Signal Processing) COE (Common Operating Environment) effort.

After I retired in 2006 I forked the TVCPP implementation to a new implementation I call JVSIP where the J is the first letter of my last name. I wanted a mechanism to continue development and support of the VSIPL effort, and I wanted interested parties to be able to access my work. TASP is long gone and funding by traditional government channels seems to have dried up. To make my work available to the community I placed JVSIP on github.

Despite a lack of traditional funding VSIPL trudges on under the guise of OMG. Also included on github are the development site for the OMG specification, and an open source implementation of VSIPL++.

Documentation is hard to do, generally lags behind implementations, and documents are always under development and seldom finished. For this reason most documentation the author does is labeled as *draft* even though I may not plan to get around to doing a *non-draft* version.

Early in 2001 the author wrote a document describing the current functionality of the TASP VSIPL Core Plus implementation called *TASP VSIPL Core Plus* which includes many examples and a fairly good overview of C VSIPL functionality. It was done in a hurry with little editing and was, of course, a draft. The document needs updating and editing but it was done originally on a Sun workstation using Framemaker; a word processing package the author liked very much. Unfortunately it was not long before Framemaker effectively died (It is still out there as part of adobe but for my purposes it is dead), the Sun workstation was replaced by a PC and Microsoft Word became the only way to do things. So the original source of the *TASP VSIPL Core Plus* book was basically lost and only the PDF document remains. Updating the document without the original tools is difficult so was never done.

I have decided re-do the previous *TASP VSIPL Core Plus* as the *JVSIP User Manual*. The contents will look a lot like the previous book but will include some editing and a lot of new information including information on how to use pyJvsip. Although this is a fresh start, since the source for the previous document is gone, the PDF content will be freely copied and pasted into this new document; many of the examples will be the same except updated and versions written in pyJvsip; and the author considers this document to be an update of the original.

The new document will be done using LaTeX on a Mac with the MacTeX distribution supplying the tools and underlying environment. LaTeX is not user friendly or wysiwyg and the author is not expert. But TeX is persistent and portable. All the necessary tools for doing a book are there and the source is all text so easily maintained.

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

VSIPL Forum - A short history

In the early 90's signal processing boards by Mercury, Sky, CSPI, and others were becoming popular for use by Government programs with compute heavy software. Each company produced their own proprietary signal processing libraries for use on their boards. This caused concern of vendor lock-in because software would need to be rewritten every time a new board procurement was done.

The TASP group was at this time trying to do a bulk contract for signal processing boards similar to the Tactical Advanced Computer (TAC) contract. In order to progress on the signal processing specification the TASP group was told they needed to specify a common operating environment for the boards.

At about this time DARPA also saw a need for a signal processing library and awarded a contract to Hughes Research Laboratory to run a forum to produce an open signal processing specification for signal and image processing; and to write a reference implementation of said specification.

TASP decided to participate in the forum as well as many other industry and university participants whose names may be found in the introductory pages to the original VSIPL document.

To make a long story short the DARPA VSIPL project and TASP have ended but the VSIPL forum has continued on in one form or another, currently with the OMG.

Success Story?

The title of this section is a question yet to be answered. Although VSIPL, as a specification, has been a minor success it may yet die for lack of support by the people who started the effort. There is only so much that volunteer efforts (such as JVSIP) can do and the cost/payback for companies developing VSIPL code is problematic without a big, paying, customer base. Lacking any sort of funding or policy by DOD to support reference implementation development, specification development, or commercial vendor buy in by VSIPL requirements in procurement documents means that the effort may yet wither on the vine.

Code History

The original code basis for the C VSIP library implementation was a very early pre-alpha (incomplete) version of the VSIPL Reference library produced by Hughes Research Laboratory of Malibu, California in December of 1997. The original HRL release was template based using m4 as a code generator. The generated library was very slow and not really suitable for writing example codes of real world problems. HRL was never successful in actually completing a complete reference implementation of VSIPL.

I was part of the TASP group and it was important to have an implementation suitable for writing real world applications. In addition at the time I did not understand m4 and, I realize now, was only marginally competent at programing in C. I copied the generated C source files from the m4 base and modified them directly. The original was slow mostly because of the method of programing. They would start with a scalar function which would be called by a general element wise function which would be called by the actual function. This was very confusing to me so I just flattened everything out so the actual function call did all the work. This produced an enormous speed-up in example codes which was what I needed.

Over time many changes were made to the TASP implementation to add performance, and to keep up with the changing VSIPL specification. I learned a lot about C programing as time went on; and some of what I learned made it into the library. Eventually the TASP implementation became a de facto reference implementation.

I suspect HRL was never successful because of a lack of funding. For a company like HRL to produce a library as extensive as VSIPL would be an expensive proposition. For various reasons I found myself with good funding and not much direction for a couple of years. The VSIPL library was similar to some codes I had wanted to write anyway; and I had just completed a masters degree at UW with signal processing as my main study. So, with no tasking from above and freedom to set my own agenda, I worked like crazy for about a year and eventually had a code base extensive enough, and well tested enough, that folks could use it.

We are approaching 20 years since the original code base and little if any of the original HRL code remains in the library. The original mostly contained support functions and simple element wise operations. Changes to the specification caused many changes to the support functions, and (as previously mentioned) the element wise functions provided by the HRL library were so slow as to be unusable for demonstration purposes. Most of what remains are the odd copyright statement. Except for (perhaps) function prototypes (defined by the spec) I would be surprised if any of the underlying code is from the original.

Contents

List of Figures

List of Tables

Chapter 1

Introduction To JVSIP

Introduction

If you are new to VSIPL and find you are confused by the various acronyms, or you find some of the terms here are unfamiliar, try reading the Preface chapter above. It contains information about the origins and meaning of JVSIP and VSIPL and may reduce the confusion factor for the person new to VSIPL.

First the big picture.

The **`JVSIP`** distribution contains a C signal processing library implementing (most of) the **`C VSIPL`** specification. **`JVSIP`** also contains a python vsip module encapsulating the C library. Once the **`vsip`** module was done a new module called **`vsiputils`** was done to provide function overloading and to reduce the name space. One of the main purposes of the **`vsiputils`** module was to help me to learn python programing; but a lot of work was done there and the module still survives although it may go away in the future. Eventually I got around to defining python classes and created the **`pyJvsip`** module. In this document we mainly treat the python interface defined in the **`pyJvsip`** module but you should be aware other modules exist.

The distribution is available on github. The distribution only contains source code. You will need a C compiler (supporting C89) to make the C Library. You will need the same C compiler, a python distribution (2.7), and a free open source package called SWIG to help encapsulate C code into python modules. The C library and the Python modules are independent except the same C source code is used for both.

Chapter one of this book will contain some basic information and an example in figure 1.1. This chapter and chapter three are for a quick start for readers who want to get started programing. Chapter two is mostly a reference chapter containing C and **`pyJvsip`** functions and usage information. Chapter four will delve more deeply into the **`block`** and **`view`** structures. Following chapters cover more complicated functions for signal processing and linear algebra.

Figure 1.1: Add Two Vectors

c VSIPL

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

#define N 6 /* the length of the vector */
void VU_vprint_f(vsip_vview_f* a){
   vsip_length i;
   for(i=0; i<vsip_vgetlength_f(a); i++)
       printf("%+.2f ",vsip_vget_f(a,i));
   printf("\n");
   return;
}
int main(){vsip_init((void*)0);
{
   vsip_vview_f *A = vsip_vcreate_f(N,0),
                *B = vsip_vcreate_f(N,0),
                *C = vsip_vcreate_f(N,0);
   vsip_randstate *rndm=\
        vsip_randcreate(7,1,1,VSIP_PRNG);
   vsip_vrandn_f(rndm,A);
   printf("A = ");VU_vprint_f(A);

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

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

   vsip_valldestroy_f(A);
   vsip_valldestroy_f(B);
   vsip_valldestroy_f(C);
   vsip_randdestroy(rndm);
   }
   vsip_finalize((void*)0);
   return 1;
}

/* output */
/*
A = -0.05 +0.59 +0.73 -0.37 -0.21 -0.83
B = +5.00 +5.00 +5.00 +5.00 +5.00 +5.00
C = A+B
C = +4.95 +5.59 +5.73 +4.63 +4.79 +4.17
*/
```

pyJvsip

```python
import pyJvsip as pv
N=6
A = pv.create('vview_f',N).randn(7)
B = A.empty.fill(5.0)
C = A.empty.fill(0.0)
print('A = '+A.mstring('%+.2f'))
print('B = '+B.mstring('%+.2f'))
pv.add(A,B,C)
print('C = A+B')
print('C = '+C.mstring('%+.2f'))
""" OUTPUT
A = [-0.05 +0.59 +0.73 -0.37 -0.21 -0.83]

B = [+5.00 +5.00 +5.00 +5.00 +5.00 +5.00]

C = A+B
C = [+4.95 +5.59 +5.73 +4.63 +4.79 +4.17]
"""
```

Polymorphism with pyJvsip

```python
import pyJvsip as pv
N=6
A = pv.create('cvview_d',N).randn(7)
B = A.empty.fill(5.0)
C = A.empty.fill(0.0)
print('A = '+A.mstring('%+.2f'))
print('B = '+B.mstring('%+.2f'))
pv.add(A,B,C)
print('C = A+B')
print('C = '+C.mstring('%+.2f'))
""" OUTPUT
A = [+0.16+0.50i -0.21-0.75i -0.56-0.09i \
     +1.15+0.45i +0.10+0.43i +0.63-1.05i]

B = [+5.00+0.00i +5.00+0.00i +5.00+0.00i \
     +5.00+0.00i +5.00+0.00i +5.00+0.00i]

C = A+B
C = [+5.16+0.50i +4.79-0.75i +4.44-0.09i \
     +6.15+0.45i +5.10+0.43i +5.63-1.05i]
"""
```

C VSIP versus pyJvsip

In this section I will make some comments about the difference between programing with c and the **C VSIP** library and programing with **pyJvsip** in the python environment.

The VSIPL library has support mechanisms for blocks and views. The **pyJvsip** module has support for blocks and views. The block and view in **pyJvsip** are instantiations of class definitions. The block and view in c VSIPL are opaque structures created with c VSIPL support functions defined for that purpose. This means a **pyJvsip** view is not the same as a VSIPL view even though I may write about them as if they are the same object. In general VSIPL objects (LUD, FFT, matrix view, etc.) created with create functions are contained inside a **pyJvsip** object as an instance variable.

The VSIPL library has a requirement for initialization and finalization. PyJvsip is written on top of the c VSIPL implementation so we still need to initialize it and finalize it. However for **pyJvsip** I have abstracted that away so that when a **pyJvsip** object is created the initialization of the object checks to see if C VSIPL has been initialized and will call **vsip_init** if it needs to. There is a special class object which keeps track of **pyJvsip** objects and when no **pyJvsip** objects are left then it calls **vsip_finalize**. So **pyJvsip** has no explicit initialization/finalization other than the required python import statement.

To avoid memory leaks there is a requirement for destruction of allocated objects after they are no longer needed in C VSIPL. For deallocation of VSIPL objects contained within a **pyJvsip** object; when a **pyJvsip** object has no reference left the python garbage collector will call the delete method. This will destroy any c VSIPL objects that have been allocated for use with the **pyJvsip** object. So **pyJvsip** has no explicit destroy functions.

Polymorphism

The encapsulation of c VSIPL using SWIG adds type information to the VSIPL python objects. Using this information, and information added to **pyJvsip** objects, as keys for python dictionaries allows VSIPL to become polymorphic. Most functions and methods in **pyJvsip** determine the underlying functionality using type information extracted from the calling object. Not every combination will necessarily work. Someplace under the covers everything must be covered. However it is generally possible to program in **pyJvsip** in a manner so that once the initial type has been chosen the rest of the code is generic even to the point of covering both real and complex.

Example

We show a simple example in figure 1.1 where we add two vectors both in C VSIPL on the left and in python on the right using **pyJvsip** .

Depth, Shape, Precision; VSIPL Naming

In order to understand `C VSIPL` one needs to understand something about the convention used when naming functions, types, structures, scalars, etc. in `C VSIPL` . This also will help one understand some of the reasons behind the **`block`** and **`view`** structures in `C VSIPL` . I try to maintain the same conventions in this document and extend them to cover **`pyJvsip`** type strings.

Depth

A scalar element has a **`depth`**. For VSIPL this is pretty simple. It is either complex or real. I suppose in the future it is possible other scalar depths could be defined. For instance a scalar defining a pixel in an image might have red, green, blue components.

Note that **`depth`** is an attribute of a **`block`**, and that **`block`**s only store elements of a single type; so a **`block`** will only have one depth associated with it.

Precision

Precision indicates how accurate the numbers are. In C this would be indicated by **`float`**, **`double`**, **`int`**, etc. `JVSIP` only supports standard ANSI C89 precisions but the naming conventions for the VSIP specification allow for just about any precision to be be declared if an implementation wants to support it.

Note that **`precision`** is an attribute of a **`block`**

Shape

A **`view`** defines the **`shape`** of a VSIPL object. A **`block`** is basically an abstract notion of memory storage. It has a **`depth`** and a **`precision`** and provides to a view a linear array of scalar elements. How the elements are defined on the underlying memory of the compute device is implementation dependent. The **`view`** then places a shape on the block allowing one to access the data as a vector, matrix, or tensor.

So the **`shape`** is an attribute of the **`view`**. Views are basically index sets.

Function Naming for `C VSIPL`

Depth Affix

Generally the prefix **`c`** is used to indicate complex and the prefix **`r`** is used to indicate real. For real the precision is frequently understood with no **`r`** except in some cases where both real and complex are needed. For instance **`vsip_vadd_f`** is for real vectors and **`vsip_cvadd_f`** is for complex vectors. The function **`add`** has been defined to allow for adding a real vector to a complex vector resulting in **`vsip_rcvadd_f`**.

We also have an **`mi`** depth type which goes in the precision place-holder. This is a matrix index type the scalar of which is defined as a structure in the **`C VSIPL`** specification (similar to the way complex is defined).

Precision Affix

There are many precisions available for use in VSIPL. The ones used in **`JVSIP`** are contained in table 1.1.

We note that the matrix index has elements that are the same precision as the vector index. The matrix index comes in the precision place when naming but it is much like the complex type and actually indicates a **`depth`**.

The type **`ue32`** is used in the definition of **`VSIPL`** random numbers. There are no blocks defined for it, only a scalar. For JVSIP this is defined to be an **`unsigned int`** which normally is 32 bits long; but I don't think this is required by the C89 specification. So in general the declaration of this type (in **`vsip.h`**) will be implementation dependent.

Table 1.1: Precision Affix in JVSIP

Precision	Affix	Comment
float	**`f`**	standard c **`float`**
double	**`d`**	standard c **`double`**
int	**`i`**	standard c signed **`int`**
short	**`si`**	standard c signed **`short int`**
unsigned char	**`uc`**	standard c **`unsigned char`**
implementation dependent	**`vi`**	Vector Index. For **`JVSIP`** **`unsigned long int`**
`C VSIPL` defined	**`mi`**	Matrix Index Actually a **`depth`**
exactly 32 bit unsigned	**`ue32`**	For **`JVSIP`** **`unsigned int`**

Shape Affix

Shapes in **`C VSIP`** library are basically indicated by an **`s`** for a scalar, a **`v`** for a vector, a **`m`** for a matrix and a **`t`** for a tensor.

Comments on Naming in C VSIPL specification

In the **`C VSIPL`** specification we have characters in italic font for ***`d`*** (depth), ***`s`*** (shape), ***`p`*** (any precision), ***`f`*** (any float), ***`i`*** (any integer), etc.

These special characters indicate an overloaded specification telling the implementor what general types may be defined for a function in an implementation. I don't use these character types in this document because I am talking about an actual implementation; not a specification. The characters I use indicate what is actually implemented.

Chapter 2

Functions

Introduction

In this chapter I give basic usage information for the functions included in the JVSIP implementation of the C VSIPL specification and also related information for the **pyJvsip** python module. There are many functions so I may miss a few.

Usage information may also be found by reading the C VSIPL specification, either the old one included with the JVSIP distribution or the newer one developed by the HPEC working group of the OMG. I currently recommend sticking with the old one included with the JVSIP distribution. There is a lot of information about C VSIPL in the specification so C VSIPL information in this document will not be extensive; and since **pyJvsip** has no specification document I will spend more time covering the **pyJvsip** methodology.

I try and include information on the **pyJvsip** methods and functions collocated with the corresponding C VSIPL information. Reading the pyJvsip.py module file is also encouraged. **PyJvsip** includes some functionality not (directly) part of C VSIPL. I will try and highlight these special cases.

For python information the python help mechanism has also been supported somewhat; but keeping that information correct, up-to-date, and available for every function is a work in progress.

Keep in mind this chapters main purpose is as a go-to reference for proper incantations when writing code. Except for the introductory sections it is probably not something you will want to read.

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 C VSIPL 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.

As discussed in chapter one python supports polymorphism, and object oriented programing. A `pyJvsip` object is an instantiation of a python class definition. The python object will contain a C VSIPL object as an instance variable as well as other information needed by `pyJvsip` . For this reason the python garbage collector will destroy C VSIPL objects when no reference to the `pyJvsip` object exists.

Because of the true object oriented nature of `pyJvsip` there are methods defined for every class which accomplish most of the functionality of C VSIPL. `PyJvsip` also defines many functions which operate on the `pyJvsip` objects. Frequently you can use either a method or a function. This information is reflected in the JVSIP function list.

No attempt is made to be exhaustive in the function descriptions. Those interested in more detail are directed to the VSIPL specification document included with the JVSIP distribution. In addition various examples included in this document will provide more detail on the use of some of the more complicated functions.

C VSIPL Specification

The main document on which `JVSIP` is based is the *VSIPL 1.3 API* as approved by the VSIPL Forum on January 31, 2008. That document is included with the `JVSIP` distribution. The main purpose of this section is to provide a roadmap for people who are familiar with the C VSIPL specification to get around in this `JVSIP` manual. Here I provide tables in an order matching the *VSIPL 1.3 API* specification with links to the same information as presented in the `JVSIP` manual.

Table 2.1: VSIPL 1.3 API Chapters

VSIPL INTRODUCTION
SUMMARY OF VSIPL TYPES
SUPPORT FUNCTIONS
SCALAR FUNCTIONS
RANDOM NUMBER GENERATION
VECTOR & ELEMENTWISE OPERATIONS
SIGNAL PROCESSING FUNCTIONS
LINEAR ALGEBRA FUNCTIONS
IMPLEMENTATION DEPENDENT INPUT AND OUTPUT
VSIPL Addendum

Summary of VSIPL Types

Support Functions

Table 2.2: Support Function Overview

Initialize and Finalize Operations

Table 2.3: Initialization

`init`	Initialize the VSIP Library
`finalize`	Finalize the VSIP Library

Block Objects

Table 2.4: Array and Block Object Functions

`blockadmit`	Admit block associated with user allocated memory.
`blockbind`	Create and bind a **C VSIPL** block to user allocated memory.
`blockcreate`	Creates a **C VSIPL** block and bind to VSIPL allocated memory.
`blockdestroy`	Free any memory allocated by **C VSIPL** associated with a block.
`blockfind`	Find the pointer to the data bound to a VSIPL released block object.
`blockrebind`	Rebind a VSIPL block to user allocated memory.
`blockrelease`	Release block associated with user allocated memory.
`complete`	Force all deferred VSIPL execution to complete.
`cstorage`	Returns the preferred complex storage format, for a precision type for this implementation.

View Objects

Table 2.5: Vector View Object Functions

alldestroy	Free both **block** and **view**
bind	Bind a **view** to a **block**
cloneview	Clone a **view**
create	Create a **view**
destroy	Free a **view**
get	Get a value from a **view**
getblock	Return **block** associated with **view**
getattrib	Get attribute structure associated with **view**
getlength	Get get length of vector **view**
getoffset	Get get offset into block of vector **view**
getstride	Get stride through block of vector**view**
imagview	Return **view** of imaginary part of complex **view**
put	Get a value from a **view**
putattrib	Set attribute structure associated with **view**
putlength	Set length of vector **view**
putoffset	Set offset into block of vector **view**
putstride	Set stride through block of vector**view**
realview	Return **view** of real part of complex **view**
subview	Create a sub-**view** of a **view**

Table 2.6: Matrix View Object Functions

alldestroy	Free both **block** and **view**
bind	Bind a **view** to a **block**
cloneview	Clone a **view**
colview	Return a column **view** (vector) of a matrix **view**
create	Create a **view**
destroy	Free a **view**
diagview	Return a diagonal **view** (vector) of a matrix **view**
get	Get a value from a **view**
getblock	Return **block** associated with **view**
getattrib	Get attribute structure associated with **view**
getcollength	Get get length of vector **view**
getrowlength	Get get length of vector **view**
getoffset	Get get offset into block of vector **view**
getcolstride	Get stride through block of vector**view**
getrowstride	Get stride through block of vector**view**
imagview	Return **view** of imaginary part of complex **view**
put	Get a value from a **view**
putattrib	Set attribute structure associated with **view**
putcollength	Set length of vector **view**
putrowlength	Set length of vector **view**
putoffset	Set offset into block of vector **view**
putcolstride	Set stride through block of vector**view**
putrowstride	Set stride through block of vector**view**
realview	Return **view** of real part of complex **view**
rowview	Return a row **view** (vector) of a matrix **view**
subview	Create a sub-**view** of a **view**
transview	Create a matrix **view** as a transpose of a matrix**view**

Table 2.7: Tensor Views

alldestroy	Free both **block** and **view**
bind	Bind a **view** to a **block**
cloneview	Clone a **view**
create	Create a **view**
destroy	Free a **view**
get	Get a value from a **view**
getattrib	Get attribute structure associated with **view**
getblock	Return **block** associated with **view**
getoffset	Get get offset into block of vector **view**
getxlength	Get get X length of tensor **view**
getxstride	Get the X stride attribute of a tensor **view**
getylength	Get get Y length of tensor **view**
getystride	Get the Y stride attribute of a tensor **view**
getzlength	Get get Z length of tensor **view**
getzstride	Get the Z stride attribute of a tensor **view**
imagview	Return **view** of imaginary part of complex **view**
matrixview	Create a matrix view of a 2-D slice of the tensor **view**
put	Get a value from a **view**
putattrib	Set attribute structure associated with **view**
putoffset	Set offset into block of vector **view**
putxlength	Set X length of tensor **view**
putxstride	Set X stride through block of tensor**view**
putylength	Set Y length of tensor **view**
putystride	Set Y stride through block of tensor**view**
putzlength	Set Z length of tensor **view**
putzstride	Set Z stride through block of tensor**view**
realview	Return **view** of real part of complex **view**
subview	Create a sub-**view** of a **view**
transview	Create a transposed of a tensor **view**
vectview	Create a vector view of a 1-D slice of the tensor **view**

Scalar Functions

In general I do not define scalar functions in **pyJvsip** . Ease of use is a major goal of the **pyJvsip** module and to further this goal I decided scalars used by or returned by **pyJvsip** functions should be normal python scalars. Using scalar functions (such as cos, sin, etc.) imported from the math module or the numpy module should work fine. That said, you can always use the C VSIPL scalar functions directly since they are in the **vsip** module which is included in the **pyJvsip** module.

Random Number Generation

Elementwise Operations

Elementwise operatons are simple operations which are done on each element in a matrix or vector. Most of the time, when more than one **`view`** is input, the **`view`** shapes will need to be the same since the operation is done to identically indexed elements for each input **`view`** and the operation result is placed in an identically indexed element of the output **`view`**.
The tables referenced in this section list elementwise operations with a link to the corresponding function page. Although the function pages are alphabetical, the lists here are in the same order (although not necessarily identical) to the order they appear in the C VSIPL specification.

Table 2.8: Vector And Elementwise Operations

Elementary Math Functions 2.9
Unary Operations 2.10
Binary Operations 2.11
Ternary Operations 2.12
Logical Operations 2.13
Selection Operations 2.14
Bitwise and Boolean Logical Operators 2.15
Element Generation and Copy 2.16
Manipulation Operations 2.17

Elementary Math

Elementary math functions constitute elementwise applications of elementary operations on **`view`**s. The term *elementary* is somewhat arbitrary but includes trigonometric functions, log functions, and exponential functions. Functions here (for elements) are defined by C 89 in the **`math.h`** header file. **`JVSIP`** generally uses this math library to do the calculations for these functions.

Table 2.9: Elementary Math Functions2.8

acos	Arccosine
asin	Arcsine
atan	Arctangent
atan2	Arctangent of Two Arguments
cos	Cosine
cosh	Hyperbolic Cosine
exp	Exponential
exp10	Exponential Base 10
log	Natural Log
log10	Base 10 Log
sin	Sine
sinh	Hyperbolic Sine
sqrt	Square Root
tan	Tangent
tanh	Hyperbolic Tangent

Unary Operations

Unary operations involve calculations on a single **view**. Functions which involve a calculation where the answer is a scalar, such as **sumval** generally have a **val** as part of the root name.

Table 2.10: Unary Operations

arg	Argument
ceil	Ceiling
conj	Conjugate
cumsum	Cumulative Sum
euler	Euler
floor	Floor
mag	Magnitude
cmagsq	Complex Magnitude Squared
meanval	Mean Value
meansqval	Mean Square Value
modulate	Modulate
neg	Negate
recip	Reciprocal
round	Round
rsqrt	reciprocal Square Root
sq	Square
sumval	Sum Value
sumsqval	Sum of Squares Value

Binary Operations

Elementwise functions requiring two inputs, either two **views** or a **view** and a scalar, are called binary operations.

Note that the table in this document is somewhat shorter than the table in the C VSIPL document. For this table, for instance, an **add** is only broken out as one function. For C VSIPL there are three function for add depending on the argument list shapes. I decided to avoid that here, partly because for **pyJvsip** I can overload the call and a single method or function name is satisfactory.

Table 2.11: Binary Operations

add	Add
div	Divide
expoavg	Exponential Average
hypot	Hypotenuse
jmul	Conjugate Multiply
mul	Multiply
vmmul	Vector Matrix Multiply
sub	Subtract

Ternary Operations

Ternary operations are those involving three inputs such as $y = a \cdot x + b$ or $y = (a + b) \cdot x$. They are defined in the element-wise chapter of the C VSIPL specification.

We note that the VSIPL specification only defines ternary operations for **views** of shape vector and precision float. Both complex and real are covered although no mixed depths are defined. Some ternary operations involve scalar constants.

Table 2.12: Ternary Operations

am	Add and multiply
ma	Multiply and add
msb	Multiply and subtract
sbm	Substract and multiply

Logical Operations

Most logical operations involve comparisons between a constant and a view, by-element; or between two **views** elementwise. Answers are either **true** or **false** and are placed elementwise in a **view** of precision **bl** of appropriate shape for the inputs.

The two exception aret the functions **alltrue** and **anytrue** which are used on **views** of precision **bl** and return a boolean **true** or **false** depending on the result of the fairly obvious question asked.

Table 2.13: Logical Operations

alltrue	All True?
anytrue	Any True?
leq	Equal?
lge	Greater than or Equal?
lgt	Greater Than?
lle	Less than or Equal?
llt	Less Than?
lne	Not Equal?

Selection Operations

Selection operations involve some logical comparison and, based upon the result, an answer is *selected* and returned; either as a scalar output (signified by **val** ending the root name), or elementwise into an appropriately sized output **view**.

Table 2.14: Selection Operations

clip	Clip
first	Find First Vector Index
invclip	Inverted Clip
indexbool	Index a Boolean **view**
max	Maximum By-Element between **views**
maxmg	Maximum Magnitude By-Element between **views**
cmaxmgsq	Maximum Magnitude Squared By-Element between complex **views**
cmaxmgsqval	Maximum Magnitude Squared Value of a complex **view**
maxmgval	Maximum Magnitude Value of a **view**
maxval	Maximum Value in a **view**
min	Minimum Elementwise between **views**
minmg	Minimum Magnitude By-Element between **views**
cminmgsq	Minimum Magnitude Squared By-Element between complex **views**
cminmgsqval	Minimum Magnitude Squared Value of a complex **view**
minmgval	Minimum Magnitude Value of a **view**
minval	Minimum Value in a **view**

Bitwise and Boolean Logical Operators

This section provides support for standard logical operators. These will operate on integer precision **views** bitwise, or on **views** of precision **bl** logically.

Table 2.15: Bitwise and Boolean Logical Operators

and	And operation
not	Not operation
or	Or operation
xor	Exclusive or operation

Element Generation and Copy

This section has functions to copy data from one place to another.

Table 2.16: Element Generation and Copy

copy	Copy **view** to **view**
copyto_user	Copy data in a **view** to user specified memory
copyfrom_user	Copy data from user specified memory to a **view**
fill	Fill a **view** with a constant value
ramp	In a vector **view** create equally space *ramp* data

Manipulation Operations

Manipulation operations are functions which copy **view**s, or parts of **view**s, from one location to another while doing some manipulation operation to convert the data. For instance the **cmplx** function takes two real **view**s and copies one **view** to the imaginary part of a complex vector and the other **view** to the real part of a complex vector.

Table 2.17: Manipulation Operations

cmplx	Complex
gather	Data Gather
imag	Imaginary Part
polar	Hypotenuse
real	Real Part
rect	Rectangular
scatter	Data Scatter
swap	Swap
binary	Not Supported
bool	Not Supported
mary	Not Supported
nary	Not Supported
serialmary	Not Supported
unary	Not Supported

Signal Processing Functions

Table 2.18: Signal Processing Functions

FFT Functions2.19
Convolution/Correlation Functions2.20
Window Functions??
Filter Functions2.22
Miscellaneous Signal Processing Functions2.23

Fast Fourier Transforms

Discrete Fourier transforms are done using an FFT algorithm. Although the VSIPL 1.3 specification has definitions for two and three dimensional FFTs **JVSIP** only supports the one dimensional version.

Table 2.19: FFT Functions 2.18

Discrete Fourier Transform Class
See function page FFT

fft	Execute FFT
fft_create	Create FFT Object
fft_setwindow	Set a window in the FFT object
fft_destroy	Free FFT object
fft_getattr	Get attributes of FFT object

Convolution and Correlation Functions

Table 2.20: Convolution and Correlation Functions 2.18

Convolution Class
See function page CONV

conv_create	Create Convolution Object
conv_destroy	Destroy Convolution Ojbect
conv_attrib	Fill attribute structure with Convolution Object Attributes
convolve	Convolve with ttbfview

Correlation Class
See function page CORR

corr_create	Create Correlation Object
corr_destroy	Destroy Correlation Object
corr_attrib	Fill attribute structure with Correlation Object Attributes
correlate	Do Correlation with `view`

Window Functions

When windows were defined in the VSIPL specification they were defined as standalone functions to create a compact block with a vector `view` and fill the view with the window coefficients. I think this was an unfortunate way to do it; we would have been better off to first create the `view` and then call a function to fill the view with the coefficients.

The method of window creation in C VSIPL makes it difficult to encapsulate windows into the `pyJvsip` methods and functions; and I don't want to create a special class just for windows. Consequently window creation has become part of the class for `pyJvsip` .

Table 2.21: Window Functions2.18

blackman	Blackman Window
cheby	Chebyshev Window
hanning	Hanning Window
kaiser	Kaiser of Window

Filter Functions

Finite Impulse Response (FIR) functions defined in C VSIPL are included in **pyJvsip** . Infinite Impulse Response functions are not implemented at this time.

Table 2.22: Filter Functions
See signal processing table 2.18

Finite Impulse Response Filter Class

See function page FIR

fir_create	Create FIR Object
fir_destroy	Free FIR Object
firflt	Filter Data
fir_getattr	Get attributes of FIR Object
fir_reset	Reset FIR Object to just created state

Miscellaneous Signal Processing Functions

Table 2.23: Miscellaneous Signal Procssing Functions2.18

histo	Historgram
freqswap	Frequency Swap

Linear Algebra Functions

Table 2.24: Linear Algebra Functions

Matrix and Vector Operations2.25
Special Linear System Solvers2.26
General Square Linear System Solver2.27
Symmetric Positive Definite Linear System Solver2.28
Over-determined Linear System Solver2.29
Singular Value Decomposition2.30

Table 2.25: Matrix and Vector Operations. 2.24

`herm`	Matrix Hermitian
`jdot`	Complex Vector Conjugate Dot Product
`gemp`	General Matrix Product
`gems`	General Matrix Sum
`kron`	Kronecker Product
`prod3`	3 by 3 Matrix Product
`prod4`	4 by 4 Matrix Product
`prod`	Matrix product
`prodh`	Matrix Hermitian Product
`jprod`	Matrix Conjugate Product
`prodt`	Matrix Transpose Product
`trans`	Matrix Transpose
`dot`	Vector Dot Product
`outer`	Vector Outer Product

Table 2.26: Special Linear System Solvers 2.24

`covsol`	Solve Covariance System
`llsqsol`	Solve Linear Least Squares Problem
`toepsol`	Solve Toeplitz System

Table 2.27: General Square Linear System Solver 2.24

LUD Function set

lud	LU Decomposition
lud_create	Create LU Decomposition Object
lud_destroy	Destroy LUD Object
lud_getattr	LUD Get Attributes
lusol	Solve General Linear System

Table 2.28: Symmetric Positive Definite (SPD) Linear System Solver 2.24

Cholesky Decomposition Function set

chold	Cholesky Decomposition
chold_create	Create Cholesky Decomposition Object
chold_destroy	Destroy CHOLD Object
chold_getattr	CHOLD Get Attributes
cholsol	Solve SPD Linear System

Table 2.29: Over-determined Linear System Solver

alldestroy	Free both **block** and **view**
bind	Bind a **view** to a **block**
cloneview	Clone a **view**
colview	Return a column **view** (vector) of a matrix **view**
create	Create a **view**
destroy	Free a **view**
diagview	Return a diagonal **view** (vector) of a matrix **view**
get	Get a value from a **view**

Table 2.30: Singular Value Decomposition

alldestroy	Free both **block** and **view**
bind	Bind a **view** to a **block**
cloneview	Clone a **view**
colview	Return a column **view** (vector) of a matrix **view**
create	Create a **view**
destroy	Free a **view**
diagview	Return a diagonal **view** (vector) of a matrix **view**

Implementation Dependent Input and Output

JVSIP does not support implementation dependent IO at this time.

VSIPL Addendum

Editing the VSIPL specification was becoming difficult because of its length and instabilities in the MS Word source document. When functions were added to the specification for interpolation, permutation and sorting they were added as separate documents in a addendum and basically glued onto the pdf. This allowed for much less editing of the MS Word source document.

VSIPL Interpolation

VSIPL Permute

VSIPL Sort

VSIP Types

This section covers the enumerated types and special structures. These are declared in the public header file **vsip.h**.

JVSIP Function List

The following pages are a list of available functions in JVSIP. The top part of each page will include a section of available C functions (basically extracted from `vsip.h`). Since the VSIPL specification is the primary source of information for C VSIPL not much more is included.
Following the list of available functions is information on how (and if) the function is supported by **`pyJvsip`** . The **`pyJvsip`** section of the function page is more extensive than the C information. Basically there is a line indicating if it is available (as a tbfview method), if it is a property, and if it is in-place. Then there is a line indicating if it is available as a **`pyJvsip`** function. Finally there is a comment section with additional information.
Note that comments may follow the C VSIPL and/or pyJvsip section and may also follow both indicating the comments pertain to the entire page and not just C or Python.

PyJvsip Methods

We note that saying the method is a property means you call it without even an empty argument list. For instance if **`a`** is a **`pyJvsip view`** then **`a.cos`** will replace the values in **`a`** with there cosine. Since it is a property we DON'T say **`a.cos()`**. Frequently, but not always, view methods are done in-place and that is also indicated. If it is not done in-place then the method will construct an appropriate output **`view`** and return it filled out with the appropriate values.
An example of a **`view`** method that is not done in place is **`copy`**. For instance **`b=a.copy`** will produce a copy of **`a`** in an appropriate view. Note the new **`b view`** will be the same precision, shape and depth as the calling **`view`** but the **`b`**lock will be of an exact size and the stride information will be the minimum stride required for the **`view`**. Additional information on copy is available on its function page.
This is the type of information included on the function pages. Since there seem to be many exceptions we won't provide a lot of rules; and instead refer to the function page.

PyJvsip Functions

In the **`pyJvsip Function`** section we provide information on function calls. For pyJvsip python function calls correspond closely with their C counterpart except that the shape, depth and precision are determined by the argument types used in the call and not by the actual name as is used by C VSIPL.

Not all C functions have a corresponding **`pyJvsip`** function call. In particular most functions that return a value will be handled using a view method with no need for a function.

add

Compute the sum of a scalar and a **view** or two **view**s. A binary operation. See table 2.11.

C VSIPL

Scalar Add Functions

```
vsip_cscalar_d vsip_cadd_d(
     vsip_cscalar_d, vsip_cscalar_d);
vsip_cscalar_d vsip_rcadd_d(
     vsip_scalar_d, vsip_cscalar_d);
vsip_cscalar_f vsip_cadd_f(
     vsip_cscalar_f, vsip_cscalar_f);
vsip_cscalar_f vsip_rcadd_f(
     vsip_scalar_f, vsip_cscalar_f);
```

Normal View - View Add Functions

```
void vsip_vadd_d(
      const vsip_vview_d*, const vsip_vview_d*,
      const vsip_vview_d*);
void vsip_vadd_f(
      const vsip_vview_f*, const vsip_vview_f*,
      const vsip_vview_f*);
void vsip_cvadd_d(
      const vsip_cvview_d*, const vsip_cvview_d*,
      const vsip_cvview_d*);
void vsip_cvadd_f(
      const vsip_cvview_f*, const vsip_cvview_f*,
      const vsip_cvview_f*);
void vsip_madd_d(
      const vsip_mview_d*, const vsip_mview_d*,
      const vsip_mview_d*);
void vsip_madd_f(
      const vsip_mview_f*, const vsip_mview_f*,
      const vsip_mview_f*);
void vsip_cmadd_d(
      const vsip_cmview_d*, const vsip_cmview_d*,
      const vsip_cmview_d*);
void vsip_cmadd_f(
      const vsip_cmview_f*, const vsip_cmview_f*,
      const vsip_cmview_f*);
void vsip_vadd_i(
      const vsip_vview_i*, const vsip_vview_i*,
      const vsip_vview_i*);
void vsip_madd_i(
      const vsip_mview_i*, const vsip_mview_i*,
      const vsip_mview_i*);
void vsip_vadd_si(
      const vsip_mview_si*, const vsip_mview_si*,
      const vsip_mview_si*);
void vsip_madd_si(
      const vsip_mview_si*, const vsip_mview_si*,
      const vsip_mview_si*);
void vsip_vadd_uc(
      const vsip_vview_uc*, const vsip_vview_uc*,
      const vsip_vview_uc*);
void vsip_vadd_vi(
      const vsip_vview_vi*, const vsip_vview_vi*,
      const vsip_vview_vi*);
```

Mixed Depth View - View Add Functions

```
void vsip_rcvadd_d(
     const vsip_vview_d*, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_rcvadd_f(
     const vsip_vview_f*, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_rcmadd_d(
     const vsip_mview_d*, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_rcmadd_f(
     const vsip_mview_f*, const vsip_cmview_f*,
     const vsip_cmview_f*);
```

Mixed Depth Scalar - View Add Functions

```
void vsip_rscvadd_d(
     vsip_scalar_d, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_rscvadd_f(
     vsip_scalar_f, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_rscmadd_d(
     vsip_scalar_d, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_rscmadd_f(
     vsip_scalar_f, const vsip_cmview_f*,
     const vsip_cmview_f*);
```

Normal Scalar - View Add Functions

```
void vsip_svadd_d(
     vsip_scalar_d, const vsip_vview_d*,
     const vsip_vview_d*);
void vsip_svadd_f(
     vsip_scalar_f, const vsip_vview_f*,
     const vsip_vview_f*);
void vsip_smadd_d(
     vsip_scalar_d, const vsip_mview_d*,
     const vsip_mview_d*);
void vsip_smadd_f(
     vsip_scalar_f, const vsip_mview_f*,
     const vsip_mview_f*);
void vsip_csvadd_d(
     vsip_cscalar_d, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_csvadd_f(
     vsip_cscalar_f, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_csmadd_d(
     vsip_cscalar_d, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_csmadd_f(
     vsip_cscalar_f, const vsip_cmview_f*,
     const vsip_cmview_f*);
void vsip_svadd_i(
     vsip_scalar_i, const vsip_vview_i*,
     const vsip_vview_i*);
void vsip_svadd_si(
     vsip_scalar_si, const vsip_vview_si*,
     const vsip_vview_si*);
void vsip_svadd_uc(
     vsip_scalar_uc, const vsip_vview_uc*,
     const vsip_vview_uc*);
void vsip_svadd_vi(
     vsip_scalar_vi, const vsip_vview_vi*,
     const vsip_vview_vi*);
```

pyjvsiph

View Method

Overloaded on plus operator.

In Place: yes

Example: `a += b; a += 2`

Elements of `view a` replaced with result.

Out of Place: yes

Example: `c = a + b; d = 2 + c`

`view c` and `view d` created and filled with result of operation.

Function

Available: yes

Example: `out = add(in1,in2,out)`

Comments

- The `add` function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned.
- This may be done in-place if `in1==out` or `in2==out`.
- Argument `in1` may be a scalar. For clues to what is allowed see C VSIPL function list.

acos

Inverse Cosine. An elementary math function. See table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_acos_f(vsip_scalar_f a);
vsip_scalar_d vsip_acos_d(vsip_scalar_d a);
void vsip_macos_d(
      const vsip_mview_d*, const vsip_mview_d*);
void vsip_macos_f(
      const vsip_mview_f*, const vsip_mview_f*);
void vsip_vacos_d(
      const vsip_vview_d*, const vsip_vview_d*);
void vsip_vacos_f(
      const vsip_vview_f*, const vsip_vview_f*);
```

pyjvsiph

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.acos`

Function

Available: yes

Example: `out = acos(in,out)`

Comments

- The **`acos`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned.
- This may be done in-place if **`in==out`**.

am

Add and multiply. An element-wise function. See ternary functions table 2.12.

C VSIPL

Available Functions

```
void vsip_cvam_d(const vsip_cvview_d*,const vsip_cvview_d*,
    const vsip_cvview_d*, const vsip_cvview_d*)
void vsip_cvam_f(const vsip_cvview_f*,const vsip_cvview_f*,
    const vsip_cvview_f*, const vsip_cvview_f*)
void vsip_cvsam_d(const vsip_cvview_d*,vsip_cscalar_d,
    const vsip_cvview_d*, const vsip_cvview_d*)
void vsip_cvsam_f(const vsip_cvview_f*,vsip_cscalar_f,
    const vsip_cvview_f*, const vsip_cvview_f*)
void vsip_vam_d(const vsip_vview_d*,
    const vsip_vview_d*,
    const vsip_vview_d*, const vsip_vview_d*)
void vsip_vam_f(const vsip_vview_f*,const vsip_vview_f*,
    const vsip_vview_f*, const vsip_vview_f*)
void vsip_vsam_d(const vsip_vview_d*,
    vsip_scalar_d,const vsip_vview_d*, const vsip_vview_d*)
void vsip_vsam_f(const vsip_vview_f*,
    vsip_scalar_f,
    const vsip_vview_f*, const vsip_vview_f*)
```

Comments

- The C VSIPL spec has separate man pages for add-multiply functions containing scalar arguments, and those containing only `view` arguments.

pyJvsip

View Method

Available: No **Property:** NA **In-Place:** NA
Example: `NA`

Function

Available: yes
Example: `out = am(in1,in2,in3,out)`

Comments

- Argument `in1` is always a `view`, argument `in2` is either a `view` or a scalar and argument `in3` is always a `view`.
- The `am` function works much the same as the C VSIPL version except that a convenience pointer to the output `view` is returned.
- This may be done in-place if an input `view` is the same as the output `view`.

arg

Compute the radian value argument of complex elements in the interval $[-\pi, \pi]$. An Unary Operation. See table 2.10

C VSIPL

Available Functions

```
vsip_scalar_d vsip_arg_d(vsip_cscalar_d);
vsip_scalar_f vsip_arg_f(vsip_cscalar_f);
void vsip_marg_d(const vsip_cmview_d*, const vsip_mview_d*);
void vsip_marg_f(const vsip_cmview_f*, const vsip_mview_f*);
void vsip_varg_d(const vsip_cvview_d*,const vsip_vview_d*);
void vsip_varg_f(const vsip_cvview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** No

Example: `out=in.arg`

Function

Available: yes

Example: `out = arg(in,out)`

Comment

- Since **`arg`** takes a view of *depth* complex and outputs to a view of *depth* real of the same *shape* and *precision* as the input view the **`arg` method** will create a view of the proper type and size and return it.
- The **`arg` function** works the same as the C VSIPL function except a convenience pointer is returned to the output view
- For the **function** limited in-place functionality exists with replacement of the real or imaginary view of the input with the output. For instance **`out=arg(in,in.realview)`** works fine.

asin

Inverse Cosine. An elementary math function. See table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_asin_f(vsip_scalar_f a);
vsip_scalar_d vsip_asin_d(vsip_scalar_d a);
void vsip_masin_d(
      const vsip_mview_d*, const vsip_mview_d*);
void vsip_masin_f(
      const vsip_mview_f*, const vsip_mview_f*);
void vsip_vasin_d(
      const vsip_vview_d*, const vsip_vview_d*);
void vsip_vasin_f(
      const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.asin`

Function

Available: yes

Example: `out = asin(in,out)`

Comments

- The **`asin`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

atan

Computes the principal radian value, $[-\pi/2, \pi/2]$, of the arctangent for each element of a **`view`**. See elementary math functions table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_atan_f(vsip_scalar_f);
vsip_scalar_d vsip_atan_d(vsip_scalar_d);
void vsip_matan_d(
      const vsip_mview_d*, const vsip_mview_d*);
void vsip_matan_f(
      const vsip_mview_f*, const vsip_mview_f*);
void vsip_vatan_d(
      const vsip_vview_d*, const vsip_vview_d*);
void vsip_vatan_f(
      const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

atan2

Arctangent of Two Arguments; An elementwise function. Computes the four quadrant radian value, $[-\pi, \pi]$, of the arctangent of the ratio of the corresponding elements of two input views. See elementary math functions table 2.9.

C VSIPL

pyJvsip

blockadmit

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockadmit` function

blockbind

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

blockcreate

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockcreate` function

blockdestroy

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

blockfind

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

blockrebind

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

blockrelease

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

complete

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

cstorage

A Block Object Support Function. See table 2.4

C VSIPL

pyJvsip

Comments

- The `blockbind` function

ceil

Ceiling. An unary operation. See table 2.10

C VSIPL

pyJvsip

Comments

- The `ceil` function is not supported in `JVSIP` at this time

Cholesky Decompostion Function Set

Cholesky Decomposition Class. **??**

C VSIPL

Available Functions

Create LU Object

```
vsip_lu_d* vsip_lud_create_d(vsip_length);
vsip_lu_f* vsip_lud_create_f(vsip_length);
vsip_clu_d* vsip_clud_create_d(vsip_length);
vsip_clu_f* vsip_clud_create_f(vsip_length);
```

Destroy LU Object

```
int vsip_lud_destroy_d(vsip_lu_d*);
int vsip_lud_destroy_f(vsip_lu_f*);
int vsip_clud_destroy_d(vsip_clu_d*);
int vsip_clud_destroy_f(vsip_clu_f*);
```

Calculate LU Decomposition

```
int vsip_lud_d(vsip_lu_d*, const vsip_mview_d*);
int vsip_lud_f(vsip_lu_f*, const vsip_mview_f*);
int vsip_clud_d(vsip_clu_d*, const vsip_cmview_d*);
int vsip_clud_f(vsip_clu_f*, const vsip_cmview_f*);
```

Solve Using Calculated LU Decomposition

```
int vsip_lusol_d(const vsip_lu_d*, vsip_mat_op,
     const vsip_mview_d*);
int vsip_lusol_f(const vsip_lu_f*, vsip_mat_op,
     const vsip_mview_f*);
int vsip_clusol_d(const vsip_clu_d*, vsip_mat_op,
     const vsip_cmview_d*);
int vsip_clusol_f(const vsip_clu_f*, vsip_mat_op,
     const vsip_cmview_f*);
```

Fill LU Attribute Structure

```
void vsip_lud_getattr_d(const vsip_lu_d*,
     vsip_lu_attr_d*);
void vsip_lud_getattr_f(const vsip_lu_f*,
     vsip_lu_attr_f*);
void vsip_clud_getattr_d(const vsip_clu_d*,
     vsip_clu_attr_d*);
void vsip_clud_getattr_f(const vsip_clu_f*,
     vsip_clu_attr_f*);
```

pyJvsip

View Methods

- A `view` method has been defined for the kernel `view`. The kernel is treated as non-symmetric so the entire kernel is assumed.[1]
- A variable argument list is supported.

  The first required argument is the input data `view`.

  The second optional argument is the decimation factor. It defaults to one.
- Other parameters are either set to there default value, or are calculated from included parameters.

In-Place: no

Out-Of-Place: yes

Example:

Finite Impulse Response Argument List

`filt`	A vector `view` of filter coefficients. Required argument
`sym`	Symmetry of `filt` kernel. Required argument
`N`	Length of input data vector. Required argument
`D`	Decimation factor. Required argument
`state`	Flag to indicate if the filter state is to be saved. `VSIP_STATE_SAVE` or `VSIP_STATE_NO_SAVE` Argument is supported but defaults to not saving. Instead of `VSIP` flags you may use the strings `'YES'` or `'NO'`.
`ntimes`	Hint for how much the LUD object will be used. Zero indicates many times. For `JVSIP` this argument is only supported at the interface level and defaults to zero.
`algHint`	Algorithm hint to optimize for speed (`VSIP_ALG_TIME`), size (`VSIP_ALG_SPACE`), or accuracy (`VSIP_ALG_NOISE`). For `JVSIP` this argument is only supported at the interface level and defaults to time.

Finite Impulse Response Filter Types

'fir_f'	Real `LUD`; float precision
'cfir_f'	Complex `LUD`; float precision
'rcfir_f'	Complex `LUD` with real `kernel`; float precision
'fir_d'	Real `LUD`; double precision
'cfir_d'	Complex `LUD`; double precision
'rcfir_d'	Complex `LUD` with real `kernel`; double precision

LUD Class Methods

For class methods table we assume we have created an LUD object we call **firObj** and we have an input **view x** compliant with **firObj** and a compliant output **view y**.

Finite Impulse Response Filter Methods

firObj.flt(x,y)	Filter the data **x** and place the results in **y**
firObj.decimation	Returns integer decimation factor.
firObj.length	Returns integer length for **x**.
firObj.lengthOut	Returns integer of valid data points in **y**
firObj.reset	Resets LUD filter to it's initial state.
firObj.state	Returns **True** if filter state is saved, otherwise returns **False**.
firObj.type	Returns string indicating filter type.
firObj.vsip	Returns C VSIPL filter instance.

- Methods **decimation**, **length**, **state**, **type** and **vsip** are set when the LUD instance is created and do not change after create
- Method **lengthOut**[2] is calculated during the execution of method **flt(x,y)** and is useful if state is saved and the filter object is used multiple times on a long piece of data.
- Method **reset** is used if state is saved and the filter is used multiple times on a long data set and then *reset*[3] to it's initial state for use on multiple long data sets.

[1] This does not preclude symmetric kernels. You just need the entire kernel.
[2] See C VSIPL specification for more information on length of output data.
[3] See signal processing text on overlap-add and overlap-save filtering.

CONV Class

Discrete Fourier Transforms. See FFT Functions table 2.19

C VSIPL

Available Functions

`fft_create`

```
vsip_rcfir_d* vsip_rcfir_create_d(
    const vsip_vview_d*, vsip_symmetry, vsip_length,
    vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
```

`fft_destroy`

```
int vsip_fft_destroy_d(vsip_fft_d*);
```

`fft`

```
int vsip_rcfirflt_d(vsip_rcfir_d*, const vsip_cvview_d*,
    const vsip_cvview_d*);
```

```
fft_getattr
   void vsip_rcfir_getattr_d(const vsip_rcfir_d*,
      vsip_rcfir_attr*);
fir_reset
   void vsip_rcfir_reset_d(vsip_rcfir_d*)
```

pyJvsip

CORR Class

Discrete Fourier Transforms. See FFT Functions table 2.19

C VSIPL

Available Functions

`fft_create`

```
vsip_rcfir_d* vsip_rcfir_create_d(
    const vsip_vview_d*, vsip_symmetry, vsip_length,
    vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
```

`fft_destroy`

```
int vsip_fft_destroy_d(vsip_fft_d*);
```

`fft`

```
int vsip_rcfirflt_d(vsip_rcfir_d*, const vsip_cvview_d*,
    const vsip_cvview_d*);
```

fft_getattr

```
void vsip_rcfir_getattr_d(const vsip_rcfir_d*,
    vsip_rcfir_attr*);
```

fir_reset

```
void vsip_rcfir_reset_d(vsip_rcfir_d*)
```

pyJvsip

cos

Cosine; An elementary math function; see table 2.9

C VSIPL

Available Functions

```
vsip_scalar_f vsip_cos_f(vsip_scalar_f a);
vsip_scalar_d vsip_cos_d(vsip_scalar_d a);
void vsip_mcos_d(
     const vsip_mview_d*, const vsip_mview_d*);
void vsip_mcos_f(
     const vsip_mview_f*, const vsip_mview_f*);
void vsip_vcos_d(
     const vsip_vview_d*, const vsip_vview_d*);
void vsip_vcos_f(
     const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.cos`

Function

Available: yes

Example: `out = cos(in,out)`

Comments

- The **`cos`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

cosh

Hyperbolic Cosine; An elementwise function

C VSIPL

pyJvsip

ceil

Ceiling. An unary operation. See table 2.10

C VSIPL

pyJvsip

Comments

- The `ceil` function is not supported in `JVSIP` at this time

conj

Conjugate each element in a view. An Unary Operations. See table 2.10

C VSIPL

Available Functions

```
vsip_cscalar_d vsip_conj_d(vsip_cscalar_d);
vsip_cscalar_f vsip_conj_f(vsip_cscalar_f);
void vsip_cmconj_d(
      const vsip_cmview_d*, const vsip_cmview_d*);
void vsip_cmconj_f(
      const vsip_cmview_f*, const vsip_cmview_f*);
void vsip_cvconj_d(
      const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvconj_f(
      const vsip_cvview_f*, const vsip_cvview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.conj`

Function

Available: yes

Example: `out = conj(in,out)`

Comments

- The **`conj`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.
- If the calling **`view`** for the **`conj`** method is real no error is generated. This case is basically a no operation. This is not true for the **`conj`** function call which will generate an assert error as an unsupported type.

covsol

Solve linear least squares problem. 2.26.

C VSIPL

Available Functions

pyJvsip

cumsum

Cumulative Sum

C VSIPL

pyJvsip

copy

Copy Data between two views. Some mixed types are supported so this method can be used to produce a copy of data of a new precision

C VSIPL

Available Functions

```
void vsip_cmcopy_d_d(
     const vsip_cmview_d*, const vsip_cmview_d*);
void vsip_cmcopy_d_f(
     const vsip_cmview_d*, const vsip_cmview_f*);
void vsip_cmcopy_f_d(
     const vsip_cmview_f*, const vsip_cmview_d*);
void vsip_cmcopy_f_f(
     const vsip_cmview_f*, const vsip_cmview_f*);
void vsip_cvcopy_d_d
     (const vsip_cvview_d*, const vsip_cvview_d*);
··· etc.
```

There are many copy functions. To see all supported search the `vsip.h` header file.[1]

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** no

Example:
```
out=in.copy
out=in.copyrm
out=in.copycm
```

The **`copy`** method creates a new view and data space that is the same shape, precision and depth as the input view and copies the data from the `in` view to the `out` view. The block in the `out` view will be the exact size needed to hold the data and will be unit stride along the major direction of the `in` view.

The **`copycm`** method is the same as the `copy` method except the output view will always be row major independent of the input views major direction.

The **`copyrm`** method is the same as the `copy` method except the output view will always be column major independent of the input views major direction.

If the input view is a vector the three copy methods have identical results.

Function

Available: yes

[1] For instance `grep copy_ vsip.h` will list all available copy functions.

Example: `out = copy(in,out)`

The **`copy`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned.

div

Divide two **view**s, a scalar and a **view** or a **view** and a scalar. A binary operation. See table 2.11.

C VSIPL

There are many combinations of divide available in **JVSIP** . The specification provides the normal **view** divides of complex-complex and real-real types; but also provides real-complex and complex-real divides as well as mixed scalar - **view** and **view**-scalar devides. Consequently the listed available functions are broken up into several tables below.

Available Functions

Scalar Functions

```
vsip_cscalar_d vsip_cdiv_d(
     vsip_cscalar_d, vsip_cscalar_d);
vsip_cscalar_d vsip_crdiv_d(
     vsip_cscalar_d, vsip_scalar_d);
vsip_cscalar_f vsip_cdiv_f(
     vsip_cscalar_f, vsip_cscalar_f);
vsip_cscalar_f vsip_crdiv_f(
     vsip_cscalar_f, vsip_scalar_f);
```

Normal View Functions

```
void vsip_vdiv_d(
     const vsip_vview_d*, const vsip_vview_d*,
     const vsip_vview_d*);
void vsip_vdiv_f(
     const vsip_vview_f*, const vsip_vview_f*,
     const vsip_vview_f*);
void vsip_mdiv_d(
     const vsip_mview_d*, const vsip_mview_d*,
     const vsip_mview_d*);
void vsip_mdiv_f(
     const vsip_mview_f*, const vsip_mview_f*,
     const vsip_mview_f*);
void vsip_cvdiv_d(
     const vsip_cvview_d*, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_cvdiv_f(
     const vsip_cvview_f*, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_cmdiv_d(
     const vsip_cmview_d*, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_cmdiv_f(
     const vsip_cmview_f*, const vsip_cmview_f*,
     const vsip_cmview_f*);
```

Mixed Depth View Functions

```
void vsip_rcvdiv_d(
     const vsip_vview_d*, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_rcvdiv_f(
     const vsip_vview_f*, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_crvdiv_d(
     const vsip_cvview_d*, const vsip_vview_d*,
     const vsip_cvview_d*);
void vsip_crvdiv_f(
     const vsip_cvview_f*, const vsip_vview_f*,
     const vsip_cvview_f*);
void vsip_rcmdiv_d(
     const vsip_mview_d*, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_rcmdiv_f(
     const vsip_mview_f*, const vsip_cmview_f*,
     const vsip_cmview_f*);
void vsip_crmdiv_d(
     const vsip_cmview_d*, const vsip_mview_d*,
     const vsip_cmview_d*);
void vsip_crmdiv_f(
     const vsip_cmview_f*, const vsip_mview_f*,
     const vsip_cmview_f*);
```

View Divide Scalar Functions

```
void vsip_vsdiv_d(
     const vsip_vview_d*, vsip_scalar_d,
     const vsip_vview_d*);
void vsip_vsdiv_f(
     const vsip_vview_f*, vsip_scalar_f,
     const vsip_vview_f*);
void vsip_cvrsdiv_d(
     const vsip_cvview_d*, vsip_scalar_d,
     const vsip_cvview_d*);
void vsip_cvrsdiv_f(
     const vsip_cvview_f*, vsip_scalar_f,
     const vsip_cvview_f*);
void vsip_msdiv_d(
     const vsip_mview_d*, vsip_scalar_d,
     const vsip_mview_d*);
void vsip_msdiv_f(
     const vsip_mview_f*, vsip_scalar_f,
     const vsip_mview_f*);
void vsip_cmrsdiv_d(
     const vsip_cmview_d*, vsip_scalar_d,
     const vsip_cmview_d*);
void vsip_cmrsdiv_f(
     const vsip_cmview_f*, vsip_scalar_f,
     const vsip_cmview_f*);
```

Scalar Divide View Functions

```
void vsip_svdiv_d(
     vsip_scalar_d, const vsip_vview_d*,
     const vsip_vview_d*);
void vsip_svdiv_f(
     vsip_scalar_f, const vsip_vview_f*,
     const vsip_vview_f*);
void vsip_rscvdiv_d(
     vsip_scalar_d, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_rscvdiv_f(
     vsip_scalar_f, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_csvdiv_d(
     vsip_cscalar_d, const vsip_cvview_d*,
     const vsip_cvview_d*);
void vsip_csvdiv_f(
     vsip_cscalar_f, const vsip_cvview_f*,
     const vsip_cvview_f*);
void vsip_smdiv_d(
     vsip_scalar_d, const vsip_mview_d*,
     const vsip_mview_d*);
void vsip_smdiv_f(
     vsip_scalar_f, const vsip_mview_f*,
     const vsip_mview_f*);
void vsip_rscmdiv_d(
     vsip_scalar_d, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_rscmdiv_f(
     vsip_scalar_f, const vsip_cmview_f*,
     const vsip_cmview_f*);
void vsip_csmdiv_d(
     vsip_cscalar_d, const vsip_cmview_d*,
     const vsip_cmview_d*);
void vsip_csmdiv_f(
     vsip_cscalar_f, const vsip_cmview_f*,
     const vsip_cmview_f*);
```

pyJvsip

View Method

Overloaded on divide operator.

In Place: yes

Example: `a /= b; a /= 2`
Elements of **`view a`** replaced with result.

Out of Place: yes

Example: `c = a / b; d = 2 / c; e = c / 2`
`view c`, **`view d`**, and **`view e`** created and filled with result of operation.

dot

Vector Dot Product 2.25.

C VSIPL

Available Functions

pyJvsip

euler

Euler

C VSIPL

pyJvsip

exp

Exponential; An elementwise function

C VSIPL

pyJvsip

exp10

Exponential Base 10; An elementwise function

C VSIPL

pyJvsip

FFT Function Set

Discrete Fourier Transforms. See FFT Functions table 2.19

C VSIPL

Available Functions

FFT Create Functions

```
vsip_fft_d* vsip_ccfftip_create_d(vsip_length, vsip_scalar_d,
   vsip_fft_dir, unsigned int, vsip_alg_hint);
vsip_fft_d* vsip_ccfftop_create_d(vsip_length,vsip_scalar_d,
   vsip_fft_dir, unsigned int, vsip_alg_hint);
vsip_fft_d* vsip_crfftop_create_d(vsip_length,vsip_scalar_d,
   unsigned int, vsip_alg_hint);
vsip_fft_d* vsip_rcfftop_create_d(vsip_length,vsip_scalar_d,
   unsigned int, vsip_alg_hint);
vsip_fft_f* vsip_ccfftip_create_f(vsip_length,vsip_scalar_f,
   vsip_fft_dir,unsigned int, vsip_alg_hint);
vsip_fft_f* vsip_ccfftop_create_f(vsip_length,vsip_scalar_f,
   vsip_fft_dir,
   unsigned int, vsip_alg_hint);
vsip_fft_f* vsip_crfftop_create_f(vsip_length,vsip_scalar_f,
   unsigned int, vsip_alg_hint);
vsip_fft_f* vsip_rcfftop_create_f(vsip_length, vsip_scalar_f,
   unsigned int, vsip_alg_hint);
```

Multiple FFT Create Functions

```
vsip_fftm_d* vsip_ccfftmip_create_d(vsip_length, vsip_length,
   vsip_scalar_d, vsip_fft_dir, vsip_major,
    unsigned int,vsip_alg_hint);
vsip_fftm_d* vsip_ccfftmop_create_d(vsip_length, vsip_length,
   vsip_scalar_d, vsip_fft_dir, vsip_major,
    unsigned int,vsip_alg_hint);
vsip_fftm_d* vsip_crfftmop_create_d(vsip_length, vsip_length,
   vsip_scalar_d, vsip_major,
    unsigned int, vsip_alg_hint);
vsip_fftm_d* vsip_rcfftmop_create_d(vsip_length, vsip_length,
   vsip_scalar_d, vsip_major,
    unsigned int, vsip_alg_hint);
vsip_fftm_f* vsip_ccfftmip_create_f(vsip_length, vsip_length,
   vsip_scalar_f, vsip_fft_dir, vsip_major,
    unsigned int,vsip_alg_hint);
vsip_fftm_f* vsip_ccfftmop_create_f(vsip_length, vsip_length,
   vsip_scalar_f, vsip_fft_dir, vsip_major,
    unsigned int,vsip_alg_hint);
vsip_fftm_f* vsip_crfftmop_create_f(vsip_length, vsip_length,
   vsip_scalar_f, vsip_major,
    unsigned int, vsip_alg_hint);
vsip_fftm_f* vsip_rcfftmop_create_f(vsip_length, vsip_length,
   vsip_scalar_f, vsip_major,
    unsigned int, vsip_alg_hint);
```

FFT Destroy Functions

```
int vsip_fft_destroy_d(vsip_fft_d*);
int vsip_fft_destroy_f(vsip_fft_f*);
```

Multiple FFT Destroy Functions

```
int vsip_fftm_destroy_d(vsip_fftm_d*);
int vsip_fftm_destroy_f(vsip_fftm_f*);
```

FFT Functions

```
void vsip_ccfftip_d(const vsip_fft_d*,
   const vsip_cvview_d*);
void vsip_ccfftip_f(const vsip_fft_f*,
    const vsip_cvview_f*);
void vsip_ccfftop_d(const vsip_fft_d*,
    const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_ccfftop_f(const vsip_fft_f*,
    const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_crfftop_d(const vsip_fft_d*,
    const vsip_cvview_d*, const vsip_vview_d*);
void vsip_crfftop_f(const vsip_fft_f*,
    const vsip_cvview_f*, const vsip_vview_f*);
void vsip_rcfftop_d(const vsip_fft_d*,
    const vsip_vview_d*, const vsip_cvview_d*);
void vsip_rcfftop_f(const vsip_fft_f*,
    const vsip_vview_f*, const vsip_cvview_f*);
```

Multiple FFT Functions

```
void vsip_ccfftmip_d(const vsip_fftm_d*,
    const vsip_cmview_d*);
void vsip_ccfftmip_f(const vsip_fftm_f*,
    const vsip_cmview_f*);
void vsip_ccfftmop_d(const vsip_fftm_d*,
    const vsip_cmview_d*, const vsip_cmview_d*);
void vsip_ccfftmop_f(const vsip_fftm_f*,
    const vsip_cmview_f*, const vsip_cmview_f*);
void vsip_crfftmop_d(const vsip_fftm_d*,
    const vsip_cmview_d*, const vsip_mview_d*);
void vsip_crfftmop_f(const vsip_fftm_f*,
    const vsip_cmview_f*, const vsip_mview_f*);
void vsip_rcfftmop_d(const vsip_fftm_d*,
    const vsip_mview_d*, const vsip_cmview_d*);
void vsip_rcfftmop_f(const vsip_fftm_f*,
    const vsip_mview_f*, const vsip_cmview_f*);
```

FFT Get Attributes Functions

```
void vsip_fft_getattr_d(
   const vsip_fft_d*, vsip_fft_attr_d*);
void vsip_fft_getattr_f(
   const vsip_fft_f*, vsip_fft_attr_f*);
```

Multiple FFT Get Attributes Functions

```
void vsip_fftm_getattr_d(
   const vsip_fftm_d*, vsip_fftm_attr_d*);
void vsip_fftm_getattr_f(
   const vsip_fftm_f*, vsip_fftm_attr_f*);
```

pyJvsip

View Methods

- Special `view` methods exist for `view`s of type float and double.
- `View` methods are defined as properties (@property) so the scale factor is always one for `view` methods.
- For out of place the method will create and return the output `view`.
- `View` methods determine if the FFT is a multiple FFT or a vector FFT by the `view` type. The `pyJvsip view major` attribute is used to determine if the multiple FFT is by row or by column.
- Out-of-place `view` methods `fftop` and `ifftop` treat real vectors as if they were complex with a zero imaginary part

In-Place: yes

Example:

Forward transform of vector `x` in-place

```
x.fftip
```

For matrix FFT multiple use major attribute.

```
x.ROW.fftip
x.COL.fftip
```

Inverse transform of vector `x` in-place

```
x.ifftip
```

Out-of-Place: yes

Example:

Real to complex and complex to real FFT

```
y=x.rcfft
z=y.crfft
```

Complex to complex transform of vector `x` out-of-place

```
y=x.fftop
```

Complex to complex inverse transform of vector **x** out-of-place

```
y=x.ifftop
```

Complex to complex multiple transform of matrix **x** out-of-place by column

```
y=x.COL.fftop
```

Complex to complex multiple transform of matrix **x** out-of-place by row

```
y=x.ROW.fftop
```

Complex to complex multiple inverse transform of matrix **x** out-of-place by column

```
y=x.COL.ifftop
```

Complex to complex multiple inverse transform of matrix **x** out-of-place by row

```
y=x.ROW.ifftop
```

FFT Class

To create an FFT object use

```
fftObj=FFT(t,*args)
```

where **args** is a tuple containing the create parameters for the FFT type selected, and **t** is a string indicating the type of FFT to create.

Note **args** will contain some or all of the following in the order listed. The exact argument list, and each type string, is shown in the Discrete Fourier Transform Class table below.

Fast Fourier Transform Argument List

`M`	Column Length
`N`	Row Length for **matrix** or Vector length for **vector**
`scl`	Scale Factor
`dir`	Direction flag for FFT either VSIP_FFT_FWD or VSIP_FFT_INV
`major`	For multiple FFT by row (VSIP_ROW) or by column (VSIP_COL)
`ntimes`	Hint for how much the FFT object will be used. Zero indicates many times
`alghint`	Algorithm hint to optimize for speed (VSIP_ALG_TIME), size (VSIP_ALG_SPACE), or accuracy (VSIP_ALG_NOISE)

Fast Fourier Transform Class Types

'ccfftip_f'	Complex-to-complex FFT float precision in-place **args = (M,N,scl,dir,ntimes,alghint)**
'ccfftop_f'	Complex-to-complex FFT float precision out-of-place **args = (M,N,scl,dir,ntimes,alghint)**
'rcfftop_f'	Real-to-complex FFT float precision out-of-place **args = (M,N,scl,ntimes,alghint)**
'crfftop_f'	Complex-to-real FFT single precision out-of-place **args = (M,N,scl,ntimes,alghint)**
'ccfftip_d'	Complex-to-complex FFT double precision in-place **args = (M,N,scl,dir,ntimes,alghint)**
'ccfftop_d'	Complex-to-complex FFT double precision out-of-place **args = (M,N,scl,dir,ntimes,alghint)**
'rcfftop_d'	Real-to-complex multiple FFT single precision out-of-place **args = (M,N,scl,ntimes,alghint)**
'crfftop_d'	Complex-to-real multiple FFT single precision out-of-place **args = (M,N,scl,ntimes,alghint)**
'ccfftmip_f'	Complex-to-complex multiple FFT single precision in-place **args = (M,N,scl,dir,major,ntimes,alghint)**
'ccfftmop_f'	Complex-to-complex multiple FFT single precision out-of-place **args = (M,N,scl,dir,major,ntimes,alghint)**
'rcfftmop_f'	Real-to-complex multiple FFT single precision out-of-place **args = (M,N,scl,major,ntimes,alghint)**
'crfftmop_f'	Complex-to-real multiple FFT single precision out-of-place **args = (M,N,major,ntimes,alghint)**
'ccfftmip_d'	Complex-to-complex multiple FFT double precision in-place **args =(M,N,scl,dir,major,ntimes,alghint)**
'ccfftmop_d'	Complex-to-complex multiple FFT double precision out-of-place **args = (M,N,scl,dir,major,ntimes,alghint)**
'rcfftmop_d'	Real-to-complex multiple FFT double precision out-of-place **args = (M,N,scl,major,ntimes,alghint)**
'crfftmop_d'	Complex-to-real multiple FFT double precision out-of-place **args = (M,N,scl,major,ntimes,alghint)**

FFT Class Methods

Below we assume we have created an FFT object we call **`fftObj`** and we have an input **`view x`** compliant with **`fftObj`** and if necessary a compliant output **`view y`**.

Fast Fourier Transform Methods

`fftObj.dft(x)`	Calculate an in-place DFT.
`fftObj.dft(x,y)`	Calculate an out-of-place DFT.
`fftObj.arg`	Return the argument list (a tuple) the FFT was created with.
`fftObj.type`	Return the FFT type (a string) the FFT was created with.
`fftObj.vsip`	Return the C VSIPL FFT Object encapsulated inside the pyJvsip FFT object.

FIR Function Set

Finite Impulse Response Class. See filter functions table
2.22

C VSIPL

Available Functions

FIR Create Functions

```
vsip_rcfir_d* vsip_rcfir_create_d(
   const vsip_vview_d*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
vsip_rcfir_f* vsip_rcfir_create_f(
   const vsip_vview_f*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
vsip_cfir_d* vsip_cfir_create_d(
   const vsip_cvview_d*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
vsip_cfir_f* vsip_cfir_create_f(
   const vsip_cvview_f*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
vsip_fir_d* vsip_fir_create_d(
   const vsip_vview_d*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
vsip_fir_f* vsip_fir_create_f(
   const vsip_vview_f*, vsip_symmetry, vsip_length,
   vsip_length, vsip_obj_state, unsigned, vsip_alg_hint);
```

FIR Destroy Functions

```
int vsip_rcfir_destroy_d(vsip_rcfir_d*);
int vsip_rcfir_destroy_f(vsip_rcfir_f*);
int vsip_cfir_destroy_d(vsip_cfir_d*);
int vsip_cfir_destroy_f(vsip_cfir_f*);
int vsip_fir_destroy_d(vsip_fir_d*);
int vsip_fir_destroy_f(vsip_fir_f*);
```

FIR Filter Functions

```
int vsip_rcfirflt_d(vsip_rcfir_d*, const vsip_cvview_d*,
   const vsip_cvview_d*);
int vsip_rcfirflt_f(vsip_rcfir_f*, const vsip_cvview_f*,
   const vsip_cvview_f*);
int vsip_cfirflt_d(vsip_cfir_d*, const vsip_cvview_d*,
   const vsip_cvview_d*);
int vsip_cfirflt_f(vsip_cfir_f*, const vsip_cvview_f*,
   const vsip_cvview_f*);
int vsip_firflt_d(vsip_fir_d*, const vsip_vview_d*,
   const vsip_vview_d*);
int vsip_firflt_f(vsip_fir_f*, const vsip_vview_f*,
   const vsip_vview_f*);
```

Get FIR Attribute Functions

```
void vsip_rcfir_getattr_d(const vsip_rcfir_d*,
   vsip_rcfir_attr*);
void vsip_rcfir_getattr_f(const vsip_rcfir_f*,
   vsip_rcfir_attr*);
void vsip_cfir_getattr_d(const vsip_cfir_d*,
   vsip_cfir_attr*);
void vsip_cfir_getattr_f(const vsip_cfir_f*,
   vsip_cfir_attr*);
void vsip_fir_getattr_d(const vsip_fir_d*,
   vsip_fir_attr*);
void vsip_fir_getattr_f(const vsip_fir_f*,
   vsip_fir_attr*);
```

Reset FIR Functions

```
void vsip_rcfir_reset_d(vsip_rcfir_d*)
void vsip_rcfir_reset_f(vsip_rcfir_f*)
void vsip_cfir_reset_d(vsip_cfir_d*)
void vsip_cfir_reset_f(vsip_cfir_f*)
void vsip_fir_reset_d(vsip_fir_d*)
void vsip_fir_reset_f(vsip_fir_f*)
```

pyJvsip

View Methods

- A **view** method has been defined for the kernel **view**. The kernel is treated as non-symmetric so the entire kernel is assumed.[1]
- A variable argument list is supported.
  The first required argument is the input data **view**.
  The second optional argument is the decimation factor. It defaults to one.
- Other parameters are either set to there default value, or are calculated from included parameters.

In-Place: no

Out-Of-Place: yes

Example:

Real FIR of real **view x** given kernel **view k** with default decimation 1.

```
y=k.firflt(x)
```

Complex FIR of complex **view x** given real kernel **view k** with user chosen decimation 3.

```
y=k.firflt(x,3)
```

Complex FIR of complex **view x** given complex kernel **view k** with default decimation 1.

```
y=k.firflt(x)
```

Note output **view y** is created and returned by the method.

FIR Class

To create an FIR object use

```
firObj=FIR(t,*args)
```

where **args** is a tuple containing the create parameters for the FIR type selected, and **t** is a string indicating the type of FIR to create.

Note **args** will contain some or all of items listed in the FIR argument list in the order listed. The type string **t** is shown in the FIR Types table below the argument list.

Finite Impulse Response Argument List

`filt`	A vector `view` of filter coefficients. Required argument
`sym`	Symmetry of `filt` kernel. Required argument
`N`	Length of input data vector. Required argument
`D`	Decimation factor. Required argument
`state`	Flag to indicate if the filter state is to be saved. `VSIP_STATE_SAVE` or `VSIP_STATE_NO_SAVE` Argument is supported but defaults to not saving. Instead of `VSIP` flags you may use the strings `'YES'` or `'NO'`.
`ntimes`	Hint for how much the FIR object will be used. Zero indicates many times. For `JVSIP` this argument is only supported at the interface level and defaults to zero.
`algHint`	Algorithm hint to optimize for speed (`VSIP_ALG_TIME`), size (`VSIP_ALG_SPACE`), or accuracy (`VSIP_ALG_NOISE`). For `JVSIP` this argument is only supported at the interface level and defaults to time.

Finite Impulse Response Filter Types

'fir_f'	Real `FIR`; float precision
'cfir_f'	Complex `FIR`; float precision
'rcfir_f'	Complex `FIR` with real `kernel`; float precision
'fir_d'	Real `FIR`; double precision
'cfir_d'	Complex `FIR`; double precision
'rcfir_d'	Complex `FIR` with real `kernel`; double precision

FIR Class Methods

For class methods table we assume we have created an FIR object we call **`firObj`** and we have an input **`view x`** compliant with **`firObj`** and a compliant output **`view y`**.

Finite Impulse Response Filter Methods

`firObj.flt(x,y)`	Filter the data **`x`** and place the results in **`y`**
`firObj.decimation`	Returns integer decimation factor.
`firObj.length`	Returns integer length for **`x`**.
`firObj.lengthOut`	Returns integer of valid data points in **`y`**
`firObj.reset`	Resets FIR filter to it's initial state.
`firObj.state`	Returns **`True`** if filter state is saved, otherwise returns **`False`**.
`firObj.type`	Returns string indicating filter type.
`firObj.vsip`	Returns C VSIPL filter instance.

- Methods **`decimation`**, **`length`**, **`state`**, **`type`** and **`vsip`** are set when the FIR instance is created and do not change after create
- Method **`lengthOut`**[2] is calculated during the execution of method **`flt(x,y)`** and is useful if state is saved and the filter object is used multiple times on a long piece of data.
- Method **`reset`** is used if state is saved and the filter is used multiple times on a long data set and then *reset*[3] to it's initial state for use on multiple long data sets.

[1]This does not preclude symmetric kernels. You just need the entire kernel.
[2]See C VSIPL specification for more information on length of output data.
[3]See signal processing text on overlap-add and overlap-save filtering.

floor

For each element in the input **`view`** round to the largest integral value not greater than the input. An unary operation. See table 2.10.

C VSIPL

pyJvsip

Comments

- The **`floor`** function is not supported in **`JVSIP`** at this time

gemp

General matrix product 2.25.

C VSIPL

Available Functions

pyJvsip

gems

General Matrix Sum 2.25.

C VSIPL

Available Functions

pyJvsip

herm

Matrix Hermitian 2.25.

C VSIPL

Available Functions

pyJvsip

jdot

Complex Vector Conjugate Dot Product 2.25.

C VSIPL

Available Functions

pyJvsip

jprod

Matrix conjugate product. 2.25.

C VSIPL

Available Functions

pyJvsip

kron

Kronecker Product 2.25.

C VSIPL

Available Functions

pyJvsip

llsqsol

Solve linear least squares problem. 2.26.

C VSIPL

Available Functions

pyJvsip

LUD Function Set

Lower-Upper Decomposition Class. 2.27

C VSIPL

Available Functions

Create LU Object

```
vsip_lu_d* vsip_lud_create_d(vsip_length);
vsip_lu_f* vsip_lud_create_f(vsip_length);
vsip_clu_d* vsip_clud_create_d(vsip_length);
vsip_clu_f* vsip_clud_create_f(vsip_length);
```

Destroy LU Object

```
int vsip_lud_destroy_d(vsip_lu_d*);
int vsip_lud_destroy_f(vsip_lu_f*);
int vsip_clud_destroy_d(vsip_clu_d*);
int vsip_clud_destroy_f(vsip_clu_f*);
```

Calculate LU Decomposition

```
int vsip_lud_d(vsip_lu_d*, const vsip_mview_d*);
int vsip_lud_f(vsip_lu_f*, const vsip_mview_f*);
int vsip_clud_d(vsip_clu_d*, const vsip_cmview_d*);
int vsip_clud_f(vsip_clu_f*, const vsip_cmview_f*);
```

Solve Using Calculated LU Decomposition

```
int vsip_lusol_d(const vsip_lu_d*, vsip_mat_op,
     const vsip_mview_d*);
int vsip_lusol_f(const vsip_lu_f*, vsip_mat_op,
     const vsip_mview_f*);
int vsip_clusol_d(const vsip_clu_d*, vsip_mat_op,
     const vsip_cmview_d*);
int vsip_clusol_f(const vsip_clu_f*, vsip_mat_op,
     const vsip_cmview_f*);
```

Fill LU Attribute Structure

```
void vsip_lud_getattr_d(const vsip_lu_d*,
     vsip_lu_attr_d*);
void vsip_lud_getattr_f(const vsip_lu_f*,
     vsip_lu_attr_f*);
void vsip_clud_getattr_d(const vsip_clu_d*,
     vsip_clu_attr_d*);
void vsip_clud_getattr_f(const vsip_clu_f*,
     vsip_clu_attr_f*);
```

pyJvsip

View Methods

- A **`view`** method has been defined for the kernel **`view`**. The kernel is treated as non-symmetric so the entire kernel is assumed.[1]
- A variable argument list is supported.
  The first required argument is the input data **`view`**.
  The second optional argument is the decimation factor. It defaults to one.
- Other parameters are either set to there default value, or are calculated from included parameters.

In-Place: no
Out-Of-Place: yes

Example:

Finite Impulse Response Argument List

`filt`	A vector `view` of filter coefficients. Required argument
`sym`	Symmetry of `filt` kernel. Required argument
`N`	Length of input data vector. Required argument
`D`	Decimation factor. Required argument
`state`	Flag to indicate if the filter state is to be saved. `VSIP_STATE_SAVE` or `VSIP_STATE_NO_SAVE` Argument is supported but defaults to not saving. Instead of `VSIP` flags you may use the strings `'YES'` or `'NO'`.
`ntimes`	Hint for how much the LUD object will be used. Zero indicates many times. For `JVSIP` this argument is only supported at the interface level and defaults to zero.
`algHint`	Algorithm hint to optimize for speed (`VSIP_ALG_TIME`), size (`VSIP_ALG_SPACE`), or accuracy (`VSIP_ALG_NOISE`). For `JVSIP` this argument is only supported at the interface level and defaults to time.

Finite Impulse Response Filter Types

'fir_f'	Real `LUD`; float precision
'cfir_f'	Complex `LUD`; float precision
'rcfir_f'	Complex `LUD` with real `kernel`; float precision
'fir_d'	Real `LUD`; double precision
'cfir_d'	Complex `LUD`; double precision
'rcfir_d'	Complex `LUD` with real `kernel`; double precision

LUD Class Methods

For class methods table we assume we have created an LUD object we call **firObj** and we have an input **view x** compliant with **firObj** and a compliant output **view y**.

Finite Impulse Response Filter Methods

`firObj.flt(x,y)`	Filter the data **x** and place the results in **y**
`firObj.decimation`	Returns integer decimation factor.
`firObj.length`	Returns integer length for **x**.
`firObj.lengthOut`	Returns integer of valid data points in **y**
`firObj.reset`	Resets LUD filter to it's initial state.
`firObj.state`	Returns **True** if filter state is saved, otherwise returns **False**.
`firObj.type`	Returns string indicating filter type.
`firObj.vsip`	Returns C VSIPL filter instance.

- Methods **decimation**, **length**, **state**, **type** and **vsip** are set when the LUD instance is created and do not change after create
- Method **lengthOut**[2] is calculated during the execution of method **flt(x,y)** and is useful if state is saved and the filter object is used multiple times on a long piece of data.
- Method **reset** is used if state is saved and the filter is used multiple times on a long data set and then *reset*[3] to it's initial state for use on multiple long data sets.

[1] This does not preclude symmetric kernels. You just need the entire kernel.
[2] See C VSIPL specification for more information on length of output data.
[3] See signal processing text on overlap-add and overlap-save filtering.

log

Natural logarithm; An element-wise function. See elementary math functions table 2.9.

C VSIPL

Available Functions

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.sin`

Function

Available: yes

Example: `out = sin(in,out)`

The **`log`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

log10

Compute the base ten logarithm; An element-wise function. See elementary math functions table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_d vsip_log10_d(vsip_scalar_d)
vsip_scalar_f vsip_log10_f(vsip_scalar_f)
void vsip_mlog10_d(
     const vsip_mview_d*, const vsip_mview_d*);
void vsip_mlog10_f(
     const vsip_mview_f*, const vsip_mview_f*);
void vsip_vlog10_d(
     const vsip_vview_d*, const vsip_vview_d*);
void vsip_vlog10_f(
     const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.sin`

Function

Available: yes

Example: `out = sin(in,out)`

The **`log10`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

ma

Multiply and add. An element-wise function. See ternary functions table 2.12.

C VSIPL

Available Functions

```
void vsip_cvma_d(const vsip_cvview_d*, const vsip_cvview_d*,
   const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvma_f(const vsip_cvview_f*, const vsip_cvview_f*,
   const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_cvsma_d(const vsip_cvview_d*, vsip_cscalar_d,
   const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvsma_f(const vsip_cvview_f*, vsip_cscalar_f,
   const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_vma_d(const vsip_vview_d*, const vsip_vview_d*,
   const vsip_vview_d*, const vsip_vview_d*);
void vsip_vma_f(const vsip_vview_f*, const vsip_vview_f*,
   const vsip_vview_f*, const vsip_vview_f*);
void vsip_vsma_d(const vsip_vview_d*, vsip_scalar_d,
   const vsip_vview_d*, const vsip_vview_d*);
void vsip_vsma_f(const vsip_vview_f*, vsip_scalar_f,
   const vsip_vview_f*, const vsip_vview_f*);
void vsip_cvsmsa_d(const vsip_cvview_d*, vsip_cscalar_d,
   vsip_cscalar_d, const vsip_cvview_d*);
void vsip_cvsmsa_f(const vsip_cvview_f*, vsip_cscalar_f,
   vsip_cscalar_f, const vsip_cvview_f*);
void vsip_vsmsa_d(const vsip_vview_d*, vsip_scalar_d,
   vsip_scalar_d, const vsip_vview_d*);
void vsip_vsmsa_f(const vsip_vview_f*, vsip_scalar_f,
   vsip_scalar_f, const vsip_vview_f*);
```

Comments

- The C VSIPL spec has separate man pages for multiply-add functions containing scalar arguments, and those containing only `view` arguments.

pyJvsip

View Method

Available: No **Property:** NA **In-Place:** NA
Example: `NA`

Function

Available: yes
Example: `out = ma(in1,in2,in3,out)`

Comments

- Argument **`in1`** is always a **`view`**, argument **`in2`** is either a **`view`** or a scalar and argument **`in3`** is either a **`view`** or a scalar.
- The **`ma`** function works much the same as the C VSIPL version except that a convenience pointer to the output **`view`** is returned.
- This may be done in-place if an input **`view`** is the same as the output **`view`**.

mag

Arctangent of Two Arguments; An elementwise function

C VSIPL

pyJvsip

magsq

Arctangent of Two Arguments; An elementwise function

C VSIPL

pyJvsip

meanval

Returns the mean value of all the elements of a view. See unary operations table 2.10

C VSIPL

Available Functions

```
vsip_cscalar_d vsip_cmmeanval_d(const vsip_cmview_d*);
vsip_cscalar_d vsip_cvmeanval_d(const vsip_cvview_d*);
vsip_cscalar_f vsip_cmmeanval_f(const vsip_cmview_f*);
vsip_cscalar_f vsip_cvmeanval_f(const vsip_cvview_f*);
vsip_scalar_d vsip_mmeanval_d(const vsip_mview_d*);
vsip_scalar_d vsip_vmeanval_d(const vsip_vview_d*);
vsip_scalar_f vsip_mmeanval_f(const vsip_mview_f*);
vsip_scalar_f vsip_vmeanval_f(const vsip_vview_f*);
```

pyJvsip

View Method

Available: Yes **Property:** Yes **In-Place:** NA

Example: `m=in.meanval`

Function

Available: No

Example: `NA`

Comments

- There seemed to be no reason to include this as a separate function for **`pyJvsip`**

meansqval

Returns the mean value of all the elements of a view. See unary operations table 2.10

C VSIPL

Available Functions

```
vsip_scalar_d vsip_cmmeansqval_d(const vsip_cmview_d*);
vsip_scalar_d vsip_cvmeansqval_d(const vsip_cvview_d*);
vsip_scalar_d vsip_mmeansqval_d(const vsip_mview_d*);
vsip_scalar_d vsip_vmeansqval_d(const vsip_vview_d*);
vsip_scalar_f vsip_cmmeansqval_f(const vsip_cmview_f*);
vsip_scalar_f vsip_cvmeansqval_f(const vsip_cvview_f*);
vsip_scalar_f vsip_mmeansqval_f(const vsip_mview_f*);
vsip_scalar_f vsip_vmeansqval_f(const vsip_vview_f*);
```

pyJvsip

View Method

Available: Yes **Property:** Yes **In-Place:** NA

Example: `msq=in.meansqval`

Function

Available: No

Example: `NA`

Comments

- There seemed to be no reason to include this as a separate function for **`pyJvsip`**

modulate

Computes the modulation of a real vector by a specified complex frequency. See unary operations table 2.10

C VSIPL

Available Functions

```
vsip_scalar_d vsip_cvmodulate_d(
    const vsip_cvview_d*, vsip_scalar_d,
    vsip_scalar_d, const vsip_cvview_d*);
vsip_scalar_d vsip_vmodulate_d(
    const vsip_vview_d*, vsip_scalar_d,
    vsip_scalar_d, const vsip_cvview_d*);
vsip_scalar_f vsip_cvmodulate_f(
    const vsip_cvview_f*, vsip_scalar_f,
    vsip_scalar_f, const vsip_cvview_f*);
vsip_scalar_f vsip_vmodulate_f(
    const vsip_vview_f*, vsip_scalar_f,
    vsip_scalar_f, const vsip_cvview_f*);
```

pyJvsip

View Method

Available: No **Property:** NA **In-Place:** NA

Example: `NA`

Function

Available: Yes

Example: `phiNew,out=modulate(in,nu,phi,out)`

Comments

- Note **`phi`** is the initial phase and the final phase is returned as **`phiNew`**. For **`pyJvsip`** we also return a convenience copy of the output vector

msb

Multiply and subtract. An element-wise function. See ternary functions table 2.12.

C VSIPL

Available Functions

```
void vsip_cvmsb_d(const vsip_cvview_d*, const vsip_cvview_d*,
   const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvmsb_f(const vsip_cvview_f*, const vsip_cvview_f*,
   const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_vmsb_d(const vsip_vview_d*, const vsip_vview_d*,
   const vsip_vview_d*, const vsip_vview_d*);
void vsip_vmsb_f(const vsip_vview_f*, const vsip_vview_f*,
   const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: No **Property:** NA **In-Place:** NA
Example: `NA`

Function

Available: Yes
Example: `out = msb(in1,in2,in3,out)`

Comments

- Arguments **`in1`**, **`in2`** and **`in3`** are always **`view`**s.
- The **`msb`** function works much the same as the C VSIPL version except that a convenience pointer to the output **`view`** is returned.
- This may be done in-place if an input **`view`** is the same as the output **`view`**.

neg

Computes the reciprocal for each element of a **view**. An elementwise function. See unary operations table 2.10

C VSIPL

Available Functions

```
vsip_cscalar_d vsip_cneg_d(vsip_cscalar_d);
vsip_cscalar_f vsip_cneg_f(vsip_cscalar_f);
void vsip_cmneg_d(
   const vsip_cmview_d*, const vsip_cmview_d*);
void vsip_cmneg_f(
   const vsip_cmview_f*, const vsip_cmview_f*);
void vsip_cvneg_d(
   const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvneg_f(
   const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_mneg_d(
   const vsip_mview_d*, const vsip_mview_d*);
void vsip_mneg_f(
   const vsip_mview_f*, const vsip_mview_f*);
void vsip_vneg_d(
   const vsip_vview_d*, const vsip_vview_d*);
void vsip_vneg_f(
   const vsip_vview_f*, const vsip_vview_f*);
void vsip_vneg_i(
   const vsip_vview_i*, const vsip_vview_i*);
void vsip_vneg_si(
   const vsip_vview_si*, const vsip_vview_si*);
```

pyJvsip

outer

Vector outer product. 2.25.

C VSIPL

Available Functions

pyJvsip

prod3

Special matrix product for 3 by 3 **views**2.25.

C VSIPL

Available Functions

pyJvsip

prod4

Special matrix product for 4 by 4 **`view`**s2.25.

C VSIPL

Available Functions

pyJvsip

prod

Matrix product. 2.25.

C VSIPL

Available Functions

pyJvsip

prodt

Matrix transpose product. 2.25.

C VSIPL

Available Functions

pyJvsip

prodh

Matrix Hermitian product. 2.25.

C VSIPL

Available Functions

pyJvsip

recip

Computes the reciprocal for each element of a **view**. An elementwise function. See unary operations table 2.10

C VSIPL

Available Functions

```
vsip_cscalar_d vsip_crecip_d(vsip_cscalar_d);
vsip_cscalar_f vsip_crecip_f(vsip_cscalar_f);
void vsip_cmrecip_d(
   const vsip_cmview_d*, const vsip_cmview_d*);
void vsip_cmrecip_f(
   const vsip_cmview_f*, const vsip_cmview_f*);
void vsip_cvrecip_d(
   const vsip_cvview_d*, const vsip_cvview_d*);
void vsip_cvrecip_f(
   const vsip_cvview_f*, const vsip_cvview_f*);
void vsip_mrecip_d(
   const vsip_mview_d*, const vsip_mview_d*);
void vsip_mrecip_f(
   const vsip_mview_f*, const vsip_mview_f*);
void vsip_vrecip_d(
   const vsip_vview_d*, const vsip_vview_d*);
void vsip_vrecip_f(
   const vsip_vview_f*, const vsip_vview_f*);
```

pyjvsiph

round

Round to nearest integral value; An elementwise function. See unary operations table 2.10

C VSIPL

pyJvsip

Comments

- The `round` function is not supported in `JVSIP` at this time

rsqrt

Reciprocal square root; An elementwise function. See unary operations table 2.10

C VSIPL

pyJvsip

sbm

Subtract and multiply. An element-wise function. See ternary functions table 2.12.

C VSIPL

Available Functions

```
void vsip_vsbm_d(const vsip_vview_d*, const vsip_vview_d*,
    const vsip_vview_d*, const vsip_vview_d*); void vsip_vsbm_f(const v
    const vsip_vview_f*, const vsip_vview_f*); void vsip_cvsbm_d(const
    const vsip_cvview_d*, const vsip_cvview_d*); void vsip_cvsbm_f(cons
    const vsip_cvview_f*, const vsip_cvview_f*);
```

pyJvsip

View Method

Available: No **Property:** NA **In-Place:** NA

Example: `NA`

Function

Available: yes

Example: **`out = sbm(in1,in2,in3,out)`**

Comments

- Arguments **`in1`**, **`in2`** and **`in3`** are always **`view`**s.
- The **`sbm`** function works much the same as the C VSIPL version except that a convenience pointer to the output **`view`** is returned.
- This may be done in-place if an input **`view`** is the same as the output **`view`**.

sin

Sine; An element-wise function. Input **`view`** elements are assumed to be in radians. See elementary math functions table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_sin_f(vsip_scalar_f a);
vsip_scalar_d vsip_sin_d(vsip_scalar_d a);
void vsip_msin_d(
     const vsip_mview_d*, const vsip_mview_d*);
void vsip_msin_f(
     const vsip_mview_f*, const vsip_mview_f*);
void vsip_vsin_d(
     const vsip_vview_d*, const vsip_vview_d*);
void vsip_vsin_f(
     const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.sin`

Function

Available: yes

Example: `out = sin(in,out)`

The **`sin`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

sinh

Hyperbolic Sine; An elementwise function. See elementary math functions table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_sinh_f(vsip_scalar_f a);
vsip_scalar_d vsip_sinh_d(vsip_scalar_d a);
void vsip_msinh_d(
     const vsip_mview_d*, const vsip_mview_d*);
void vsip_msinh_f(
     const vsip_mview_f*, const vsip_mview_f*);
void vsip_vsinh_d(
     const vsip_vview_d*, const vsip_vview_d*);
void vsip_vsinh_f(
     const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** yes

Example: `inOut.sinh`

Function

Available: yes

Example: `out = sinh(in,out)`

The **`sinh`** function works much the same as the C VSIPL version except that a convenience pointer to the output view is returned. This may be done in-place if **`in==out`**.

sqrt

Square Root; An elementwise function. See elementary math functions table 2.9.

C VSIPL

pyJvsip

sq

Square each element in a **`view`**. See unary operations table 2.10

C VSIPL

pyJvsip

sumval

Returns the sum of the the elements of a **`view`**. Does not modify input. See unary operations table 2.10.

C VSIPL

Available Functions

```
vsip_cscalar_d vsip_cmsumval_d(const vsip_cmview_d*);
vsip_cscalar_d vsip_cvsumval_d(const vsip_cvview_d*);
vsip_cscalar_f vsip_cmsumval_f(const vsip_cmview_f*);
vsip_cscalar_f vsip_cvsumval_f(const vsip_cvview_f*);
vsip_scalar_d vsip_msumval_d(const vsip_mview_d*);
vsip_scalar_d vsip_vsumval_d(const vsip_vview_d*);
vsip_scalar_f vsip_msumval_f(const vsip_mview_f*);
vsip_scalar_f vsip_vsumval_f(const vsip_vview_f*);
vsip_scalar_i vsip_vsumval_i(const vsip_vview_i*);
vsip_scalar_si vsip_vsumval_si(const vsip_vview_si*);
vsip_scalar_uc vsip_vsumval_uc(const vsip_vview_uc*);
vsip_scalar_vi vsip_msumval_bl(const vsip_mview_bl*);
vsip_scalar_vi vsip_vsumval_bl(const vsip_vview_bl*);
```

pyJvsip

sumsqval

Returns the sum of the squares of all the elements of a **view**. Does not modify input. See table 2.10

C VSIPL

Available Functions

```
vsip_scalar_d vsip_msumsqval_d(const vsip_mview_d* );
vsip_scalar_d vsip_vsumsqval_d(const vsip_vview_d* );
vsip_scalar_f vsip_msumsqval_f(const vsip_mview_f* );
vsip_scalar_f vsip_vsumsqval_f(const vsip_vview_f* );
```

pyJvsip

View Method

Available: yes **Property:** yes **In-Place:** NA

Example: `aValue=in.sumsqval`

Function

Available: No

Example:

Comments

- Since the **sumsqval** function returns a scalar without modifying the **view** there seemed little point in supporting this as a separate function call for **pyJvsip** .

tan

Tangent; An elementwise function. Input `view` elements are assumed to be in radians. See elementary math functions table 2.9.

C VSIPL

pyJvsip

tanh

Hyperbolic Tangent; An elementwise function. See elementary math functions table 2.9.

C VSIPL

Available Functions

```
vsip_scalar_f vsip_tanh_f(vsip_scalar_f);
vsip_scalar_d vsip_tanh_d(vsip_scalar_d);
void vsip_mtanh_d(
     const vsip_mview_d*, const vsip_mview_d*);
void vsip_mtanh_f(
     const vsip_mview_f*, const vsip_mview_f*);
void vsip_vtanh_d(
     const vsip_vview_d*, const vsip_vview_d*);
void vsip_vtanh_f(
     const vsip_vview_f*, const vsip_vview_f*);
```

pyJvsip

toepsol

Solve Toeplitz system. 2.26.

C VSIPL

Available Functions

pyJvsip

trans

Matrix transpose. 2.25.

C VSIPL

Available Functions

pyJvsip

Chapter 3

Introduction to JVSIP Programming

Introduction

Support Functions

Block Creation

Vector Creation

Other methods of view creation and view modification

Viewing the Real and Imaginary portions of a Complex Vector

VSIPL Input and Output Methods

Chapter 4

Blocks and Views

Introduction

Block Fundamentals

View Fundamentals

Vectors and vector manipulation

Matrices and matrix manipulation

Chapter 5

Introduction Boolean, Gather, Scatter and Indexbool

Introduction

Chapter 6

Signal Processing

Introduction

c VSIPL Example 1

```c
#include<vsip.h>
#define ripple 100
#define Nlength 101
int main(){vsip_init((void*)0);
{
     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);
     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);
     /*printf("CW = "); VU_vprintm_f("%6.8f ;\n",Cw); */
     VU_vfprintyg_f("%6.8f\n",Cw,"Cheby_Window");
     vsip_cvfill_f(vsip_cmplx_f(0,0),FCW);
     { vsip_vview_f *rv = vsip_vrealview_f(FCW);
       vsip_vcopy_f_f(Cw,rv);
       vsip_ccfftip_f(fft,FCW);
       vsip_vcmagsq_f(FCW,rv);
       { vsip_index ind;
         vsip_scalar_f max = vsip_vmaxval_f(rv,&ind);
         vsip_scalar_f min = max/(10e12);
         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);
     } vsip_finalize((void*)0); return 0;
}
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;
}
```

Fourier Transforms

Vector FFT

(

Vector FFT by Row or Column

Convolution, Correlation and FIR Filtering

Window Creation

VSIPL provides functions to create Blackman, Chebyshev, Hanning and Kaiser windows. Unlike most functions in C 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.

For pyJvsip the windows are defined as a method on a view so the functionality, from a user perspective, is to create a vector of a certain type and length and then fill the vector with a window. Size information are taken from the calling view. Under the covers the C VSIPL window functions are used so a copy is actually taking place to meet the requirements of pyJvsip.

Miscellaneous

Histogram

Data Reorganization

Frequency Swapping

Chapter 7

Linear Algebra

Introduction

VSIPL specifies support for standard matrix operations such as matrix products, methods to solve the standard matrix equation $A\vec{x} = \vec{b}$, 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

Simple Solvers

LU Decomposition

Cholesky Decompostion

QR Decompostion

Singular Value Decomposition