VSIPL Python Module

Version 0.51

March 12, 2012

Introduction

This document describes an encapsulation of a public domain C VSIPL library as a python module. This document is intended to be a python starting point for folks who download the distribution. It is not intended as a standalone document; but is intended to be used with the distribution.

I use the publicly available simple wrapper and interterface generator tool (SWIG Version 2.0.4; http://www.swig.org/), along with a C VSIPL library I write and maintain, to create the module. The distribution this document comes in supplies both the library C code and an interface file for SWIG (called **`vsip.i`**) which is used with SWIG to produce a **`vsip.py`** file and a **`vsip_wrap.c`** file. I also include a setup.py file to allow installation of the module into a python environment.

My expectation for the vsip module (which may or may not correspond to what really happens) are folks writing C VSIPL library routines will be able to use the python interactive environment for writing and debugging the C VSIPL code. Since python acts like a *main*, then the VSIPL objects are persistent in a python interactive shell, and the VSIPL objects can be examined interactively using other readily available python modules (like matplotlib for instance). In light of this, for this particular module, I will try and keep the differences between the C version and the Python version to the minimum required by the language differences.

Also note that although the entire C libary is made into a module the usage of functions like copy to and from user space, block bind (to user memory) and admit, release are not very helpfull in a python environment without additional work in the C domain.

Notation

In this document C code will be yellow, and python code will be green to avoid confusion. Short sections of code in python look very like their C counterpart.

How to build module

Assuming you download the distribution using git into a directory

```
$HOME/vsip
```

From a command line (designated with >) do:

```
> swig -python vsip.i
```

I also supply a setup.py file which may be used to build the module in the standard way.

```
> python setup.py install
```

I am assuming that necessary development tools (like an ANSI C compiler, python, and SWIG) have been installed on the platform.

If necessary permissions for the platform are not available one can do

```
> python setup.py install --user
```

to install to a users environment.

Python VSIP Fundamentals

Using the default build will result in a module called *vsip*. Since C VSIPL function names are not likely to be the same as anything else I generally import them without a name space.

If the vsip module is imported as

```
from vsip import *
```

then using the VSIPL module is pretty much the same as in C. If you are in an interactive environment then you can do (for instance)

```
a = vsip_vcreate_f(100,VSIP_MEM_NONE)
```

to create a VSIPL float vector of length 100. If you then do (for instance)

```
vsip_vramp_f(0,1,a)
for i in range(10):
    vsip_vget_f(a,i)
```

You get

```
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
```

The main difference between C and python is in python you don't have to typedef your variables before you assign them. Also python likes to return results. Two results would be returned as a tuple for instance. In VSIPL if more than one result is returned then the extra is handled as an argument which is passed by reference. In addition most VSIPL functions, such as the ramp above, operate by modifying the data in the vector or matrix passed in as one of the arguments and the functions return void. This seems to work well in python (for C VSIPL anyway); however it is not very python like.

Note there are other differences between C and python. In particular python does not handle pointers very well. There are ways to work around this with some additional C code; however, with one caveat described below for vector index, I have chosen not to create any additional functionality for now.

Since python does not (directly) allow low level memory access functions that require a pointer to memory like admit, release, and copy from/to user space are not very useful in python (AFAIK; I am still learning python).

Some functions in VSIPL allow passing a NULL pointer or a pointer to (void *)0 which is generally considered to be equivalent to passing a NULL pointer. For python this is equivalent to the None argument. So for instance to initialize the library in C you do

```
vsip_init((void*)0);
```

in python this is

```
vsip_init(None)
```

For another example VSIPL defines a complex scalar type in the standard way as

```
typedef struct {
      vsip_scalar_f r,i;
} vsip_cscalar_f;
```

To get a complex scalar in C you do

```
vsip_scalar_f a_cscalar;
```

which would be allocated at compile time or you could do

```
vsip_scalar_f * a_cscalar_p;
a_cscalar_p= (vsip_scalar_f*) malloc(sizeof(vsip_cscalar_f));
```

to allocate a complex scalar at run time.

Assuming the former you would use `a_scalar` like

```
a_scalar = vsip_cvget_f(a,i);
```

to get a complex scalar from vector `a` at index `i` or

```
vsip_cvput_f(a,i,&a_scalar);
```

to put a complex scalar into position at index `i` for vector `a`.

So this was problematic when I first went to try the VSIP python module. How had swig handled these cases? How do I create a complex scalar?

Turns out that in python you do

```
a_cscalar = vsip_cscalar_f
```

To get the equivalent to the C complex scalar. The real part is `a_cscalar.r` and the imaginary part is `a_scalar.i`. This is the same as C.

The python `a_scalar` also acts like a pointer when passed as a VSIPL pointer. So the put operation above becomes

```
vsip_cvput_f(a,i,a_scalar)
```

This is not exactly the same as C, but close. All VSIPL typedefs that are based on structures (vsip_vattr_d,vsip_lu_attr_f, vsip_scalar_mi, etc.) operate in the same fashion.

Simple defined data types from the vsip header file which SWIG does not handle specially act like regular scalars. For instance `vsip_scalar_f`, `vsip_index`, `vsip_length`, etc. translate into the regular python scalar types of float and int. If you do

```
a_cscalar = vsip_scalar_f
```

Python will complain that `vsip_scalar_f` is not defined. However

```
a_scalar=vsip_vget_f(a,i)
```

or

```
a_scalar = vsip_vget_d(a,i)
```

will both just return a python float type.

This is fine as long as the mechanism used in C is a regular variable. But there is no simple python mechanism (AFAIK) to convert these to a pointer to scalar. This causes a problem in some of the selection operations. For instance

```
vsip_index i,
vsip_vview_f *a;
vsip_scalar_f afloat;
/* ...some code  ... */
afloat = vsip_vmaxval_f(a,&i);
```

This returns the max value in vector `a` in scalar `afloat` and the index where the value was found into index `i`. Although I found a way around this in the swig methodology the result of that was to have these functions return a tuple. I wanted a more VSIPL like solution for this interface. So I created 3 functions in C code (file *indexptr.c*) that define an index pointer, a method to return the index pointer to python, and a method to free the memory associated with the index pointer. So an example in python using this method you do

```
>>>from vsip import *
>>> vsip_vramp_f(0,1,a)
>>> vsip_vput_f(a,4,100)
>>> i=vindexptr()
>>> m=vsip_vmaxval_f(a,i)
>>> indx=vindexptrToInt(i)
>>> vindexfree(i)
>>> m
100.0
>>> indx
4
```

The three new functions are vindexptr, vindexptrToInt, and vindexfree. Usage should be obvious from the code snippet.

Note that vsip_vmaxval_f may be called with a null pointer in C if only the max value is required and the index is not needed. In python this would look like

```
m=vsip_vmaxval_f(a,None)
```

Also note it is easy to examine a in this example by copying the data to a list using (something like)

```
>>> ["%4.1f" % vsip_vget_f(a,i) for i in
range(vsip_vgetlength_f(a))]
[' 0.0', ' 1.0', ' 2.0', ' 3.0', '100.0', ' 5.0', ' 6.0', '
7.0', ' 8.0', ' 9.0']
```

Example

In python, using the supplied C VSIPL, it is easy to forget (or ignore) C VSIPL requirements for init, finalize, and destroy. For this example I will try to be conformant to VSIPL requirements. Note that a memory leak in python is still a memory leak; and if a commercial vendor offers up a module similar to mine that his implementation may depend upon initialization for proper operation.

This example is the simple vector add (A Simple First Example) in the TASP VSIPL Core Plus book (pdf document in doc directory of distribution

Example 1. Python code for vsip vector add

```
1  from vsip import *
2  N=8  #length of vector
3  init = vsip_init(None)
4  def VU_vprint_f(a):
5      print(" ".join( ["%4.0f" % vsip_vget_f(a,i) for i in
6  range(vsip_vgetlength_f(a))]))
7
8  A = vsip_vcreate_f(N,0)
9  B = vsip_vcreate_f(N,0)
10 C = vsip_vcreate_f(N,0)
11 vsip_vramp_f(0,1,A)
12 print("A = \n"); VU_vprint_f(A)
13 vsip_vfill_f(5,B)
14 print("B = \n"); VU_vprint_f(B)
15 vsip_vadd_f(A,B,C)
16 print("C = \n"); VU_vprint_f(C)
17 vsip_valldestroy_f(A)
18 vsip_valldestroy_f(B)
19 vsip_valldestroy_f(C)
20 vsip_finalize(None)
```

for the Python Example

```
> python example1.py

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

Example 1. C code for vector add

```
1  #include<stdio.h>
2  #include<vsip.h>
3  #define N 8 /* the length of the vector */
4
5
6  void VU_vprint_d(vsip_vview_d* a){
7    vsip_length i;
8    for(i=0; i<vsip_vgetlength_d(a); i++)
9    printf("%4.0f",vsip_vget_d(a,i));
10   printf("\n");
11 }
12
13
14 int main(){
15   vsip_init((void*)0);
16   vsip_vview_d *A = vsip_vcreate_d(N,0), *B = vsip_vcreate_d(N,0),
17                  *C = vsip_vcreate_d(N,0);
18   vsip_vramp_d(0,1,A);
19   printf("A = \n");VU_vprint_d(A);
20   vsip_vfill_d(5,B);
21   printf("B = \n");VU_vprint_d(B);
22   vsip_vadd_d(A,B,C);
23   printf("C = \n");VU_vprint_d(C);
24   vsip_valldestroy_d(A);
25   vsip_valldestroy_d(B);
26   vsip_valldestroy_d(C);
27   vsip_finalize((void*)0);
28   return 1;
29 }
30
31
```

for the C example

```
> cc -o example1 example1.c -L$HOME/local/lib -I$HOME/local/include
-lvsip
> ./example1
A =
   0    1    2    3    4    5    6    7
B =
   5    5    5    5    5    5    5    5
C =
   5    6    7    8    9   10   11   12
```