pyJvsip Module User Manual

Version 0.15

July 5, 2012

DRAFT

Introduction

This document is a user manual for pyJvsip. Module pyJvsip is still in its inception. Eventually, if I stay at it long enough, I will write a specification. For now this is just a user manual for a proof of concept. The interface may change.

Note I don't have time to write a very good user manual; so I will include a lot of examples and hope people can pick stuff up as they use it. Feel free to examine the pyJvsip.py file to see how things work and to fix bugs which I am sure there are many. Ditto for the vsiputils.py file.

The goal of pyJvsip is to provide VSIPL functionality in a python environment. Note that ease of use is the primary goal for pyJvsip. It is expected that this version will be used, for the most part, interactively as an aid to prototyping C VSIPL and understanding VSIP methodology.

Notation

In this document C code will be yellow highlighted, and python code will be green to avoid confusion.

How to build module

This module requires the vsip module and the vsiputils module to be installed.

To build go to jvsip/python/pyJvsip directory in a terminal and enter

```
python setup.py install
```

pyJvsip Fundamentals

I have defined python classes using C VSIPL created objects under the covers. For this reason much of pyJvsip will be limited by the C VSIPL library. Even though we have this limitation some functionality may be extended either by python code, or by using C extensions.

Initialize and Finalize, Create and Destroy

In C VSIPL you must start with an `vsip_init`, do some creates, calculate for awhile and when done destroy all created objects, and end with a `vsip_finalize`. The design of pyJvsip means we only have creates when a class instantiates a new object. No init, finalize, or destroy should be needed (assuming I have designed the module properly).

When creating an object you must supply a supported type as a string to the create function. A class method exists which will return supported types (see **supported** below).

Knowledge of C VSIPL naming conventions will allow one to decipher what type of object will be created. For instance

```
b=Block('block_d',100)
```

will create a block **b** of size 100 elements of type real with double precision and

```
c=Block('cblock_f',50)
```

will create a block **c** of size 50 elements of type complex with float precision.

In C VSIPL this would look like (for the first case above)

```
vsip_block_d *b = vsip_blockcreate_d(100,VSIP_MEM_NONE);
```

Many C VSIPL hints will not be exposed in the pyJvsip implementation so in the case above the VSIPL memory hint is not exposed. After the initial create the type information will, for the most part, disappear from the API. You still need to pay attention to the types so you can query pyJvsip objects using the parameter method

```
anObject.type
```

which will return a string indicating the type. For instance (for the first case above)

```
b.type
```

would return string 'block_d'.

Environment

There are various environments under which python can run, and various versions of python to run. For this document I am using **Python 2.7.2** under **IPython 0.14.dev.** I start with a terminal window in OSX Lion and use the command

```
> ipython --pylab
```

If this command works properly then the modules numpy and matplotlib have been loaded. To load pyJvsip I generally do

```
import pyJvsip as pv
```

To load vsiputils.py I generally do

```
import vsiputils as vsip
```

and finally to load vsip.py I do

```
from vsip import *
```

So if you see something like

```
a=pv.create('vview_d',10).fill(0)
```

then you know I am working from the pyJvsip module and if you see

```
vsip.ramp(0,1,a.view)
```

you will know the vsiputils module is loaded and if you see

```
vsip_vsumval_d(a.view)
```

you will know the vsip module is loaded.

Block

To create a block object use

```
aBlock = pv.Block(blockType,blockLength)
```

where:

`blockType`
: is a string indicating the type of block.

  Valid strings are

  'block_f', 'block_d', 'cblock_f', 'cblock_d',

  'block_i', 'block_si', 'block_uc', 'block_bl',

  'block_vi','block_mi'

`blockLength`
: is an integer indicating the number of elements the block can store.

Methods

supported

A class method which returns a dictionary of supported types for blocks and views.

Example

```
In [22]: import pyJvsip as pv
In [23]: aDict = pv.Block.supported()
In [24]: blocks = aDict['blockTypes']
In [25]: views = aDict['viewTypes']
In [26]: blocks # some reformatting below to save space
Out[26]:
['block_f', 'block_d', 'cblock_f', 'cblock_d', 'block_vi',
 'block_mi' 'block_bl', 'block_si', 'block_i', 'block_uc']
```

bind

This method creates a view and binds it to the block. There are other functions and ways to create a view but all views must be associated with a block. The block associated with a view is immutable. There may be many views on a block but only one block in a view. A view is an index set. This function creates a view object and sets the initial set of indices.

The index set is a tuple. This may change in the future.

Example

```
# This example is verbose for clarity.
@ Probably not the way you would do it
In [1]: import pyJvsip as pv
In [2]: aBlock=pv.Block('block_d',200)
In [3]: vectorOffset = 10
```

```
In [4]: vectorStride = 2
In [5]: vectorLength = 4
In [6]: vectorTuple = (vectorOffset,vectorStride,vectorLength)
In [7]: aVector=aBlock.bind(vectorTuple)
In [8]: matrixOffset = 20
In [9]: rowStride = 1
In [10]: colStride = 5
In [11]: rowLength = 4
In [12]: colLength = 4
In [13]: matrixTuple = (matrixOffset,colStride,colLength,row-
Stride,rowLength)
In [14]: aMatrix = aBlock.bind(matrixTuple)
In [16]: aVector.ramp(0,1)

Out[16]: <pyJvsip.__View at 0x106127950>
In [17]: aVector.list
Out[17]: [0.0, 1.0, 2.0, 3.0]

In [18]: aMatrix.fill(1.5)
Out[18]: <pyJvsip.__View at 0x106127ad0>

In [21]: aMatrix.mprint('%.1f')
[ 1.5  1.5  1.5  1.5;
  1.5  1.5  1.5  1.5;
  1.5  1.5  1.5  1.5;
  1.5  1.5  1.5  1.5]
```

copy

Property method to create a new block of the same size and type. Data in the block is NOT copied.

Example

```
In [23]: anotherBlock = aBlock.copy
In [24]: len(aBlock)
Out[24]: 200

In [26]: len(anotherBlock)
Out[26]: 200

In [27]: aBlock.type
Out[27]: 'block_d'

In [28]: anotherBlock.type
Out[28]: 'block_d'
```

View

The canonical method for creating a view is to use bind method from the block it is associated with. There are also convenience functions included with pyJvsip which will create a vector or matrix directly. These *create* functions will create a block of size sufficient to hold the views data and then create view. The block associated with a view is returned with the view method (property) block. So for instance

```
import pyJvsip as pv
aView = pv.Block('block_d',10).bind((0,1,10))
```

is the same as

```
aView = pv.create('vview_d',10)
```

If you want a view of every other element of aView starting at index 1 you could then do

```
anotherView = aView.block.bind((1,2,5))
```

View Basics

There are many ways to do the above.

Example

```
In [1]: import pyJvsip as pv
In [2]: a=pv.Block('block_d',10).bind((0,1,10)).fill(0.0)
In [3]: b=a.block.bind((1,2,5)).ramp(1,1)
In [4]: b.mprint('%.1f')
[ 1.0  2.0  3.0  4.0  5.0]

In [5]: a.mprint('%.1f')
[ 0.0  1.0  0.0  2.0  0.0  3.0  0.0  4.0  0.0  5.0]
```

The above example shows how if two views associated with the same block they access the same data.

There is a copy method defined on views. This method will create a new block, attach a compliant view, and copy the data from the creating view into the new view. Note these data are independent. Also note the new block and view are exact fits. When a block and view exactly fit with minimum strides for each dimension we say the view is compact. In the example above vector a is compact. It has unit stride and length of 10 so the index space encompasses the entire block. Vector b has stride 2 so is not compact.

Example

```
# continued from above
In [6]: c=b.copy
In [7]: c.mprint('%.1f')
[ 1.0  2.0  3.0  4.0  5.0]

In [8]: c.ramp(2,2) #returns convenience reference
Out[8]: <pyJvsip.__View at 0x105727890>
```

```
In [9]: c.mprint('%.1f')
[ 2.0  4.0  6.0  8.0  10.0]
In [10]: b.mprint('%.1f')
[ 1.0  2.0  3.0  4.0  5.0]
```

For the above example `c` is compact. It is an index set on a new block of length 5 of the same type as `b.block`. It has unit stride.

Example

```
# continued from above
In [15]: b.block == c.block
Out[15]: False
In [16]: b.stride
Out[16]: 2
In [17]: c.stride
Out[17]: 1
In [18]: len(b.block)
Out[18]: 10
In [19]: len(c.block)
Out[19]: 5
In [20]: c.length == b.length
Out[20]: True
```

There is also something called a clone. A clone is a new view that is identical index set to the creating view. Note the data space is the same for the clone. Clones are useful for performance reasons and to help when a new index set on the same data is needed. (You may want to review the first example above to see how `a` and `b` were initialized.)

Example

```
# continued from above
In [21]: d=b.clone
In [22]: d.block == b.block
Out[22]: True

In [23]: d.stride
Out[23]: 2

In [24]: d.offset
Out[24]: 1

In [25]: d.putoffset(0)
Out[25]: <pyJvsip.__View at 0x101118810>
```

Example 1. C code for vector add

```c
#include<stdio.h>
#include<vsip.h>
#define N 8 /* the length of the vector */

void VU_vprint_d(vsip_vview_d* a){
  vsip_length i;
  for(i=0; i<vsip_vgetlength_d(a); i++)
  printf("%4.0f",vsip_vget_d(a,i));
  printf("\n");
}

int main(){
  vsip_init((void*)0);
  vsip_vview_d *A = vsip_vcreate_d(N,0), *B = vsip_vcreate_d(N,0),
                 *C = vsip_vcreate_d(N,0);
  vsip_vramp_d(0,1,A);
  printf("A = \n");VU_vprint_d(A);
  vsip_vfill_d(5,B);
  printf("B = \n");VU_vprint_d(B);
  vsip_vadd_d(A,B,C);
  printf("C = \n");VU_vprint_d(C);
  vsip_valldestroy_d(A);
  vsip_valldestroy_d(B);
  vsip_valldestroy_d(C);
  vsip_finalize((void*)0);
  return 1;
}
```

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
```

```
In [26]: a.mprint('%.1f');b.mprint('%.1f');d.mprint('%.1f')
[ 0.0  1.0  0.0  2.0  0.0  3.0  0.0  4.0  0.0  5.0]
[ 1.0  2.0  3.0  4.0  5.0]
[ 0.0  0.0  0.0  0.0  0.0]
```

Some final points before leaving this introduction. A *view* and a *block* are real objects that must be allocated. These are structures in C. The block also has memory associated with it where data is stored. The block may be thought of as a hidden index set which abstracts system memory into contiguous VSIPL data types. The view then puts an index set on the block which allows the data to be viewed as a vector or a matrix. So clone and copy methods, or any of the many methods which need to allocate memory, are somewhat processor intensive. Also note the clone method is fairly lightweight compared to copy, and reseting attributes in an already available view is very lightweight involving no allocation.

Given the object oriented nature of the pyJvsip module create and destroy are happening a lot in the background; but it is convenient when developing routines. If performance becomes an issue you may want to consider methods for early binding and late destroys.

Also note doing something like (continuing the code above)

```
In [27]: f=d
In [28]: f
Out[28]: <pyJvsip.__View at 0x101118810>

In [29]: d
Out[29]: <pyJvsip.__View at 0x101118810>

In [32]: f==d
Out[32]: True

So f and d are the same object. No new view was created. But
in 13 above a new view was created so

In [33]: f==b
Out[33]: False
```

Using VSIPL view contained in pyJvsip View object

You may extract the VSIPL view from the view object and use it with any appropriate *vsip* or *vsiputils* function.

Note you should not change these views in the View objects or destroy them. This would be an error. Modifying the data accessed by the view is fine.

Example

```
In [34]: import vsiputils as vsip
In [35]: attr=vsip.getattrib(b.view)
In [36]: (attr.offset,attr.stride,attr.length)
Out[36]: (1, 2, 5)
```

```
In [37]: len=vsip.getlength(b.view)
In [38]: [vsip.get(b.view,i) for i in range(len)]
Out[38]: [1.0, 2.0, 3.0, 4.0, 5.0]

In [45]: vsip.mul(2.0,b.view,b.view)
Out[45]: <Swig Object of type 'vsip_vview_d *' at 0x104367870>

In [46]: [vsip.get(b.view,i) for i in range(len)]
Out[46]: [2.0, 4.0, 6.0, 8.0, 10.0]

In [47]: b.mprint('%.1f')
[ 2.0  4.0  6.0  8.0  10.0]
```

Note in the example above b is an instantiation of the View class and has methods associated with it. The property b.view returns the VSIPL view object contained by the View object which may be used with *vsip* or *vsiputils* functions.

Important

If there is no reference to the VSIPL view object then the python garbage collector will destroy the object and any VSIPL view contained in the object. If you use a VSIPL view from a View object and the view object has been destroyed then there is a segfault which can cause python to act abnormally. For this reason you should only use the view directly when it is contained within a pyJvsip View object which has a reference defined.

In the example below `aMatrix` is the reference to a View matrix object. The matrix is printed using a function defined to use VSIPL style views. We then try the same thing without first retaining the View object.

Example

```
#good

In [1]: import pyJvsip as pv
In [2]: import vsipUser as vu
In [3]: aMatrix =
pv.Block('block_d',12).bind((0,1,3,3,4)).fill(1)

In [5]: vu.mprint(aMatrix.view,'%.1f')
[1.0 1.0 1.0 1.0;
 1.0 1.0 1.0 1.0;
 1.0 1.0 1.0 1.0]

#bad

In [6]
vu.mprint(pv.Block('block_d',12).bind((0,1,3,3,4)).fill(1).vie
w,'%.1f')
```

Line [6] above caused (for this case; symptoms seem to be non-deterministic) my python shell to freeze up.

Note that

```
In [1]: import pyJvsip as pv
In [2]:
pv.Block('block_d',12).bind((0,1,3,3,4)).fill(1).mprint('%.1f'
)
[ 1.0  1.0  1.0  1.0;
  1.0  1.0  1.0  1.0;
  1.0  1.0  1.0  1.0]
```

Works fine.

Chaining

In the examples above there are lines of code where several methods are chained together. There can sometimes be unintended consequences to this so some care is needed. In addition, although chaining makes it easy to write code, there may be performance penalties.

For an example you frequently need to initialize a view and it is handy to do that at the same time you create the view. So for instance

```
aMatrix = pv.create('mview_d',3,3,'ROW').fill(0)
```

might be used. Lets say what you really want is an identity matrix so you do

```
aMatrix =
pv.create('mview_d',3,3,'ROW').fill(0).diagview(0).fill(1)
```

The second case is in error. What happens in the first case is you create a matrix and initialize it with zeros and return a reference to the matrix in `aMatrix`. In the second case you create a matrix, initialize it with zeros, create a vector view along the main diagonal, fill the main diagonal with 1, and then return a reference to the vector along the main diagonal. The matrix is garbage collected. So you wanted an identity matrix but you got a vector of ones.

The proper way is:

```
aMatrix = pv.create('mview_d',3,3,'ROW').fill(0)

aMatrix.diagview(0).fill(1)
```

What happens above is you create and return a matrix of zeroes in the first line and the second line finds the main diagonal and fills it with zero. The second line also returns a vector view of the main diagonal but since you are not interested in it you don't create a reference so python will garbage collect the vector view.

Support Functions

Support functions allow one to manipulate views by creating new views from available views or by modifying view attributes. Also copy functionality is considered to be a support function.

Simple Math

Operations *=, +=, /=, -=

These are done in-place. Views must be of the same type although scalar operations are also allowed.

Example

```
In [1]: import pyJvsip as pv
In [2]: v1=pv.create('vview_d',5).fill(0)
In [3]: v2=v1.copy
In [4]: v1.ramp(1,1); v2.ramp(.1,.1)
Out[4]: <pyJvsip.__View at 0x1051aa810>

In [5]: v1.mprint('%.1f');v2.mprint('%.2f')
[ 1.0  2.0  3.0  4.0  5.0]
[ 0.10  0.20  0.30  0.40  0.50]
In [6]: v1 *= v2
In [7]: v1.mprint('%.1f');v2.mprint('%.2f')
[ 0.1  0.4  0.9  1.6  2.5]
[ 0.10  0.20  0.30  0.40  0.50]

In [8]: v2 *= 10
In [9]: v2.mprint('%.2f')
[ 1.00  2.00  3.00  4.00  5.00]
In [10]: v2 += 10 * v1
In [11]: v2.mprint('%.2f');v1.mprint('%.2f')
[ 2.00  6.00  12.00  20.00  30.00]
[ 0.10  0.40  0.90  1.60  2.50]
```

Let examine this a little. It is important to understand what is going on here. In lines 2 and 3 we create two vectors. They are defined on separate blocks but they are the same type and size so they are compatible for operations.

In line 4 we place some data in them using the ramp function and print out the result in line 5

In line 6 we multiply, element-wise, the elements in vector `v2` times the elements in vector v1 and place the result in `v1`.

In line 8 we multiply, element-wise, the scalar 10 time each element in vector v2 and place the result in vector `v2`.

In line 10 things become a little more interesting. Here we multiply the scalar 10 times the vector `v1`. The result is then multiplied element-wise times the vector `v2` and the result is placed into `v2`. To do this a temporary vector is created underneath the covers where the result of `10 * v1` is placed. This vector is then added to `v2` and when done there is no reference to the temporary vector so it is garbage collected. This is handy; but a performance hit.

You could also do

```
v1 *=10; v2+=v1;
```

This is not as clear but no temporary needs to be created; although the values in `v1` are altered.

A higher performance way would be to use the vector-scalar multiply, vector add function from the vsip module. This is not convenient but once your code is done and working and you need performance it may be a way to go. Currently this function is not include with pyJvsip although it may be in the future. This would look like

```
In [13]: from vsip import *
In [14]: vsip_vsma_d(v1.view,10.0,v2.view,v2.view)
```

This is also a way to get an out of place operation if the output vector were replaced with an independent vector.

Operations *, /, +, -

These are similar to the operations in the previous section but they are done out of place. A new (compact) view is created to handle the output data. In the example below I create a script and execute it from python shell.

Example

```
In [18]: cat ex1.py

import pyJvsip as pv
v1 = pv.Block('block_d',20).bind((0,1,10)).ramp(1,1)
v2 = v1.block.bind((10,1,10)).ramp(2,2)
v3 = v1 + v2
v4 = v1 * v3
v5 = v1 * v3 + v3*2.5 + v1/v3
v3.mprint('%3.1f')
v4.mprint('%3.1f')
v5.mprint('%3.1f')
In [19]: execfile('ex1.py')
[ 3.0  6.0  9.0  12.0  15.0  18.0  21.0  24.0  27.0  30.0]
[ 3.0  12.0  27.0  48.0  75.0  108.0  147.0  192.0  243.0
300.0]
[ 10.8  27.3  49.8  78.3  112.8  153.3  199.8  252.3  310.8
375.3]
```

Operations sin, cos, etc.

Simple math operations defined as methods are generally properties and are done in place. For instance for (float) vector `a` we might do `a.sin` to do an element-wise sin on each element and then replace the element with the value. To do this out of place do `aResult = a.copy.sin`.

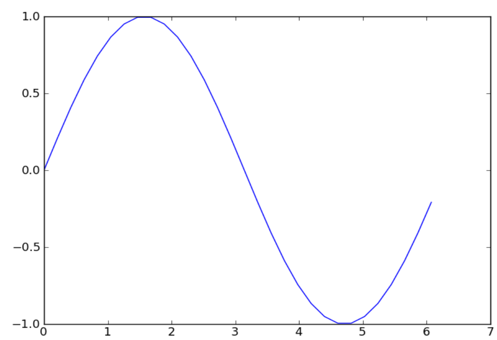
Example

Operations *, /, +, -

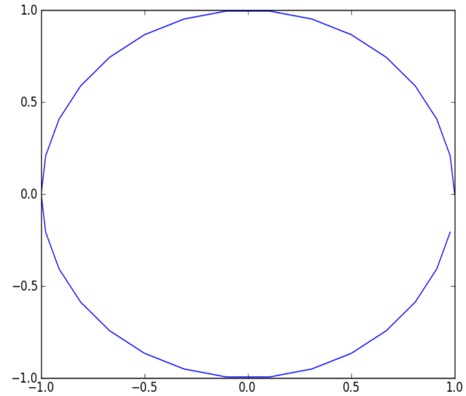
These are similar to the operations in the previous section but they are done out of place. A new (compact) view is created to handle the output data. In the example below I create a script and execute it from python shell.

Example

```
In [21]: from numpy import pi as pi
In [22]: x = pv.vector('vview_d',30).ramp(0,pi/15.0)
In [23]: y = x.copy.sin
In [24]: plot(x.list,y.list)
```




```
In [25]: x=x.cos
In [26]: plot(x.list,y.list)
```



Unintended Consequences

~~I tried doing line 25 above as just `x.cos` (not `x=x.cos`) but this resulted in cos being applied twice. I don't understand what is going on here. It seems to be some interaction between a trig function available in numpy which is automatically loaded in the pylab environment I am operating in; but I don't have a clue why (or how) it should operate on my view.~~

update

The problem with things being done twice seems to have been a bug in ipython 0.12 dev and some sort of interaction with the @property decorator. I have updated to ipython 0.14 dev and things seem to be working better now.

Discrete Fourier Transform

JVSIP supports methods for computing DFT's. There is an FFT class for instantiating an FFT object for a particular kind and size of view. The FFT object can also be created with the create function. In addition view methods are supplied for doing DFT operations. This section describes this functionality.

FFT Class

The key to the argument list for the FFT instantiation using the Class is in the VSIP C specification. Usage is:

```
fftObj=FFT(t,arg)
```

where `t` is a type corresponding to one of

```
['ccfftip_f','ccfftop_f', 'rcfftop_f','crfftop_f',
 'ccfftip_d', 'ccfftop_d', 'rcfftop_d', 'crfftop_d',
 'ccfftmip_f', 'ccfftmop_f','rcfftmop_f', 'crfftmop_f',
 'ccfftmip_d', 'ccfftmop_d', 'rcfftmop_d', 'crfftmop_d']
```

and `arg` is a tuple corresponding to the argument list of the specific FFT specified with the type argument.

Note that C VSIPL has enumerated types `vsip_fft_dir`, `vsip_alg_hint` and for FFT multiple `vsip_major`. These are needed when creating an FFT object with the Class.

FFT Methods

arg

This is a property which returns the tuple used to create the FFT object.

dft

Returns the discrete Fourier transform of the calling fft object. The input is either a single view if the FFT object was created for in-place operations, or two compliant views if out of place. See VSIP specification for information on compliant views for `rcfftmop` and `crfftmop`.

fft

This is a property and returns the VSIPL FFT object contained in the JVSIP FFT object. This object should not be set or destroyed using VSIPL functionality; it is owned by the JVSIP FFT object.

type

This is a property and returns a string indicating the FFT type.

Example

In the example below we create an example function which we call with some length. It does an FFT on the ramp and an inverse FFT on the result; in-place.

FFT Example Using FFT Class

```
import pyJvsip as pv
def atest(n):
    scale = 1.0
    dir = pv.VSIP_FFT_FWD
    hint = pv.VSIP_ALG_TIME
    ntimes=1
    dftLength=n
    arg=(dftLength, scale, dir, ntimes,hint)
    fftObj=pv.FFT('ccfftip_f',arg)
    v=pv.create('cvview_f',dftLength).ramp(0,1)
    v.mprint('%.1f')
    fftObj.dft(v)
    v.mprint('%.3f')
    dir=pv.VSIP_FFT_INV
    scale = 1.0/float(dftLength)
    arg=(dftLength, scale, dir, ntimes,hint)
    ifftObj=pv.FFT('ccfftip_f',arg)
    ifftObj.dft(v)
    v.mprint('%.1f')
In [23]: atest(10)
[ 0.0+0.0i  1.0+0.0i  2.0+0.0i  3.0+0.0i  4.0+0.0i
5.0+0.0i  6.0+0.0i  7.0+0.0i  8.0+0.0i  9.0+0.0i]
[ 45.000+0.000i -5.000+15.388i -5.000+6.882i
-5.000+3.633i -5.000+1.625i -5.000+0.000i -5.000-1.625i
-5.000-3.633i -5.000-6.882i -5.000-15.388i]
[ 0.0-0.0i  1.0+0.0i  2.0-0.0i  3.0-0.0i  4.0-0.0i
5.0+0.0i  6.0-0.0i  7.0-0.0i  8.0+0.0i  9.0+0.0i]
```

Create Function and the FFT

The create function may be used to create an FFT object.

Note that when the class is used to create the FFT object you must fully populate the argument tuple. This can be a little painful and usually involves a review of the manual to understand the argument list.

The create function is designed to be more user friendly than the class and will select some default values if the user does not provide them. The first argument is the **type string** for the FFT object. It is required. The second (and third for FFTM) argument is the **size** of the FFT, and it is also required. A default **scale** value of 1.0 is supplied by the create function. If a scale is supplied then it must follow the size argument. The FFT object, by default, will do a forward FFT. If an inverse FFT is required then the string 'INV' should be supplied following the size arguments and any scale argument if it is supplied. For the multiple FFT the direction of the FFT should be supplied as 'COL' or 'ROW' for column or row direction. If this argument is not supplied it will perform the FFT by row (default).

Note the algorithm hints for performance include an integer (`ntimes`) as to how often the object will be used, and a `vsip_alg_hint` for performance tuning. The JVSIP VSIP implementation does not support these hints (ignores the arguments) so the create function sets `ntimes` to zero (a lot of times) and the `vsip_alg_hint` is always `VSIP_ALG_TIME` (fastest). If the python code is used with some other VSIPL implementation one may want to support these hints but pyJvsip create function will not at this time.

FFT Example Using create function

```
import pyJvsip as pv
def atest(n):
    dftLength=n
    fftObj=pv.create('ccfftip_f',dftLength)
    v=pv.create('cvview_f',dftLength).ramp(0,1)
    v.mprint('%.1f')
    fftObj.dft(v)
    v.mprint('%.3f')
    scale = 1.0/float(dftLength)
    ifftObj=pv.create('ccfftip_f',dftLength,scale,'INV')
    ifftObj.dft(v)
    v.mprint('%.1f')
In [23]: atest(10)
[ 0.0+0.0i  1.0+0.0i  2.0+0.0i  3.0+0.0i  4.0+0.0i
5.0+0.0i  6.0+0.0i  7.0+0.0i  8.0+0.0i  9.0+0.0i]
[ 45.000+0.000i -5.000+15.388i -5.000+6.882i
-5.000+3.633i -5.000+1.625i -5.000+0.000i -5.000-1.625i
-5.000-3.633i -5.000-6.882i -5.000-15.388i]
[ 0.0-0.0i  1.0+0.0i  2.0-0.0i  3.0-0.0i  4.0-0.0i
5.0+0.0i  6.0-0.0i  7.0-0.0i  8.0+0.0i  9.0+0.0i]
```

FFT View Methods

Views support VSIPL type methods for FFT as properties. These are very easy to use because they have no options (they are a property). Inverse FFT is handled by defining both forward and inverse properties. Out of place versions create a view of the proper type and return the data in that. There are performance issues when using these; but for the purpose of interactive programing they should be time savers.

fftip, ifftip

Preform, in-place, an FFT on a complex, float, vector or matrix. The multiple FFT direction (if a matrix calls this) is set by the major flag. Default is by row. Scaling is set to 1.0 for both forward and inverse versions; so use an element-wise scalar-view multiply to scale.

FFT Example Using view in-place method

```
import pyJvsip as pv
def atest(n):
    v=pv.create('cvview_f',n).ramp(0,1)
    v.mprint('%.1f')
    v.fftip
    v.mprint('%.3f')
    v.ifftip
    v.mprint('%.1f')
    v *= 1.0/float(n)
    v.mprint('%.1f')
In [35]: atest(10)
[ 0.0+0.0i  1.0+0.0i  2.0+0.0i  3.0+0.0i  4.0+0.0i
5.0+0.0i  6.0+0.0i  7.0+0.0i  8.0+0.0i  9.0+0.0i]
[ 45.000+0.000i -5.000+15.388i -5.000+6.882i
-5.000+3.633i -5.000+1.625i -5.000+0.000i -5.000-1.625i
-5.000-3.633i -5.000-6.882i -5.000-15.388i]
[ 0.0-0.0i  10.0+0.0i  20.0-0.0i  30.0-0.0i  40.0-0.0i
50.0+0.0i  60.0-0.0i  70.0-0.0i  80.0+0.0i  90.0+0.0i]
[ 0.0-0.0i  1.0+0.0i  2.0-0.0i  3.0-0.0i  4.0-0.0i
5.0+0.0i  6.0-0.0i  7.0-0.0i  8.0+0.0i  9.0+0.0i]
```

fftop, ifftop

Preform, out-of-place, an FFT on a complex, float, vector or matrix. The multiple FFT direction (if a matrix calls this) is set by the major flag. Default is by row. Scaling is set to 1.0 for both forward and inverse versions; so use an element-wise scalar-view multiply to scale. The calling view is not modified. A new view of the proper size and type is created and returned by the method.

FFT Example Using view out-of-place method

```
import pyJvsip as pv
def atest(n):
    v=pv.create('cvview_f',n).ramp(0,1)
    v1 = v.fftop
    v2 = 1.0/float(n) * v1.ifftop
    v.mprint('%.1f')
    v1.mprint('%.3f')
    v2.mprint('%.1f')
In [35]: atest(10)
[ 0.0+0.0i  1.0+0.0i  2.0+0.0i  3.0+0.0i  4.0+0.0i
5.0+0.0i  6.0+0.0i  7.0+0.0i  8.0+0.0i  9.0+0.0i]
[ 45.000+0.000i -5.000+15.388i -5.000+6.882i
-5.000+3.633i -5.000+1.625i -5.000+0.000i -5.000-1.625i
-5.000-3.633i -5.000-6.882i -5.000-15.388i]
[ 0.0-0.0i  1.0+0.0i  2.0-0.0i  3.0-0.0i  4.0-0.0i
5.0+0.0i  6.0-0.0i  7.0-0.0i  8.0+0.0i  9.0+0.0i]
```

rcfft, crfft

Preform, out-of-place, real to complex or complex to real FFT on a float vector or matrix. The multiple FFT direction (if a matrix calls this) is set by the major flag. Default is by row. Scaling is set to 1.0 for both real to complex and complex to real versions; so use an element-wise scalar-view multiply to scale. The calling view is not modified. A new view of the proper size and type is created and returned by the method.

FFT Example Using rcfft/crfft method

```
import pyJvsip as pv
def atest(n):
    v=pv.create('vview_f',n).ramp(0,1)
    v1 = v.rcfft
    v2 = 1.0/float(n) * v1.crfft
    v.mprint('%.1f')
    v1.mprint('%.3f')
    v2.mprint('%.1f')
In [44]: atest(10)
[ 0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0]
[ 45.000+0.000i -5.000+15.388i -5.000+6.882i
-5.000+3.633i -5.000+1.625i -5.000+0.000i]
[-0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0]
```