C Extensions to NumPy and Python

Draft version 0.1. 12/7/06

by Lou Pecora

Overview

Introduction– a little background

In my use of Python I came across a typical problem: I needed to speed up particular parts of my code. I am not a Python guru or any kind of coding/computer guru. I use Python for numerical calculations and I make heavy use of Numeric/NumPy. Almost every Python book or tutorial tells you build C extensions to Python when you need a routine to run fast. C extensions are C code that can be compiled and linked to a shared library that can be imported like any Python module and you can call specified C routines like they were Python functions.

Sounds nice, but I had reservations. It looked non-trivial (it is, to an extent). So I searched for other solutions. I found them. They are such approaches as SWIG (http://www.swig.org/), Pyrex (http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/), ctypes (http://python.net/crew/theller/ctypes/), Psycho (http://psyco.sourceforge.net/), and Weave (http://www.scipy.org/Weave). I often got the simple examples given to work (not all, however) when I tried these. But I hit a barrier when I tried to apply them to NumPy. Then one gets into typemaps or other hybrid constructs. I am not knocking these approaches, but I could never figure them out and get going on my own code despite lots of online tutorials and helpful suggestions from various Python support groups and emailing lists.

So I decided to see if I could just write my own C extensions. I got help in the form of some simple C extension examples for using Numeric written about 2000 from Tom Loredo of Cornell university. These sat on my hard drive until 5 years later out of desperation I pulled them out and using his examples, I was able to quickly put together several C extensions that (at least for me) handle all of the cases (so far) where I want a speedup. These cases mostly involve passing Python integers, floats (=C doubles), strings, and NumPy 1D and 2D float and integer arrays. I rarely need to pass anything else to a C routine to do a calculation. If you are in the same situation as me, then this package I put together might help you. It turns out to be fairly easy once you get going.

Please note, Tom Loredo is not responsible for any errors in my code or instructions although I am deeply indebted to him. Likewise, this code is for research only. It was tested by only my development and usage. It is not guaranteed, and comes with no warranty. Do not use this code where there are any threats of loss of life, limb, property, or money or anything you or others hold dear.

I developed these C extensions and their Python wrappers on a Macintosh G4 laptop using system OS X 10.4 (essential BSD Unix), Python 2.4, NumPy 0.9x, and the gnu compiler and linker gcc. I think most of what I tell you here will be easily translated to Linux and other Unix systems beyond the Mac. I am not sure about Windows. I hope that my low-level approach will make it easy for Windows users, too.

The code (both C and Python) for the extensions may look like a lot, but it is **very** repetitious. Once you get the main scheme for one extension function you will see that pattern repeated over and over again in all the others with minor variations to handle different arguments or return different objects to the calling routine. Don't be put off by the code. The good news is that for many numerical uses extensions will follow the same format so you can quickly reuse what you already have written for new projects. Focus on one extension function and follow it in detail (in fact, I will do this below). Once you understand it, the other routines will be almost obvious. The same is true of the several utility functions that come with the package. They help you create, test, and manipulate the data and they also have a lot of repetition. The utility functions are also very short and simple so nothing to fear there.

General Scheme for NumPy Extensions

This will be covered in detail below, but first I wanted to give you a sense of how each extension is organized.

Three things that must be done before your C extension functions in the C source file.

• You must include Python and NumPy headers.

• Each extension must be named in a defining structure at the beginning of the file. This is a name used to access the extension from a Python function.

• Next an initialization set of calls is made to set up the Python and NumPy calls and interface. It will be the same for all extensions involving NumPy and Python unless you add extensions to access other Python packages or classes beyond NumPy arrays. I will not cover any of that here (because I don't know it). So the init calls can be copied to each extension file.

Each C extension will have the following form.

• The arguments will always be the same: `(PyObject *self, PyObject *args)`
Don't worry if you don't know what exactly these are. They are pointers to general Python objects and they are automatically provided by the header files you will use from NumPy and Python itself. You need know no more than that.

• The args get processed by a function call that parses them and assigns them to C defined objects.

• Next the results of that parse might be checked by a utility routine that reaches into the structure representing the object and makes sure the data is the right kind (float or int, 1D or 2D array, etc.). Although I included some of these C-level checks, you will see that I think they are better done in Python functions that are used to wrap the C extensions. They are also a lot easier to do in Python. I have plenty of data checks in my calling Python wrappers. Usually this does not lead to much overhead since you are not calling these extensions billions of times in some loop, but using them as a portal to a C or C++ routine to do a long, complex, repetitive, specialized calculation.

• After some possible data checks, C data types are initialized to point to the data part of the NumPy arrays with the help of utility functions.

• Next dimension information is extracted so you know the number of columns, rows, vector dimensions, etc.

• Now you can use the C arrays to manipulate the data in the NumPy arrays. The C arrays and C data from the above parse point to the original Python/NumPy data so changes you make affect the array values when you go back to Python after the extension returns. You can pass the arrays to other C functions that do calculations, etc. Just remember you are operating on the original NumPy matrices and vectors.

• After your calculation you have to free any memory allocated in the construction of your C data for the NumPy arrays. This is done again by Utility functions. This step is only necessary if you allocated memory to handle the arrays (e.g. in the matrix routines), but is not necessary if you have not allocated memory (e.g. in the vector routines).

• Finally, you return to the Python calling function, by returning a Python value or NumPy array. I have C extensions which show examples of both.

Python Wrapper Functions

It is best to call the C extensions by calling a Python function that then calls the extension. This is called a Python wrapper function. It puts a more pythonic look to your code (e.g. you can use keywords easily) and, as I pointed out above, allows you to easily check that the function arguments and data are correct before you hand them over to the C extension and other C functions for that big calculation. It may seem like an unnecessary extra step, but it's worth it.

Contiguous Arrays

This may seem like an arcane subject to inject at this point, but it is important and understanding it will help you avoid problems and bugs that are difficult to solve and find. C arrays must be contiguous (FORTRAN has the same requirement although the data order is by columns unlike C's order by rows). Contiguous means all the data is in one sequential block of memory. It is possible in Python to generate non-contiguous data in an array. For example,

```
c=arange(16)
d=c[::2]
```

The array `d` points to every other component of `c`. If you change a value of `d` you change the corresponding value of the `c` array, too:

```
print c
d[0]=132
print c
print d
```

yields the output,

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[132   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15]
[132   2   4   6   8  10  12  14]
```

If you pass the array `d` to a C extension and try to convert it to a C array you will have trouble and it may be hard to debug. You need to convert d to a contiguous data array. This can be done in Python or in the C extension. I show both examples, below. However, I recommend you test for contiguity in Python

before trying to pass the array and if it is not contiguous you change it there (I show you how below). If you do so, it will become a new array, no longer point to `c`, i.e. it will have it own data block, and it will be contiguous. Then you can safely pass it to the C extension. If you generate a contiguous array in the C code instead, it seems it will not point back to the original array so you cannot operate on that array in place. And you might run into problems with garbage collection since C versions of Python objects are not automatically dereferenced when the function exits. There are no problems like this if you handle the conversion of arrays to contiguous form first in Python. The whole topic of DECREF and INCREF in C extensions is an entire book chapter by itself. If you can stay away from having to use those functions, do so. With one exception I do that here and advise you to do the same. If all you are doing is passing arrays, that should not be a problem. Prepare them in Python first to be contiguous.

The Code

In this section I refer to the code in the source files `C_arraytest.h`, `C_arraytest.c`, `C_arraytest.py`, and `C_arraytest.mak`. You should keep those files handy (probably printed out) so you can follow the explanations of the code below. The souce files are also listed in the Appendix.

The C Code – one detailed example with utilities.

First, I will use the example of code from `C_arraytest.h`, `C_arraytest.c` for the routine called `matsq`. This function takes a (NumPy) matrix *A*, integer *i*, and (Python) float *y* as input and outputs a return (NumPy) matrix *B* each of whose components is equal to the square of the input matrix component times the integer times the float. Mathematically, :

$$B_{ij} = i\, y\, (A_{ij})^2$$

The Python code to call the `matsq` routine is,

```
A=matsq(B,i,y).
```

Here is the relevant code in one place:

The Header file, `C_arraytest.h`:

```
/* ==== Prototypes ================================== */

// ... skip other prototypes
static PyObject *matsq(PyObject *self, PyObject *args);

/* .... C matrix utility functions .................*/
PyArrayObject *pymatrix(PyObject *objin);
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin);
double **ptrvector(long n);
void free_Carrayptrs(double **v);
int  not_doublematrix(PyArrayObject *mat);
```

The Source file, `C_arraytest.c`:

```
#include "Python.h"
#include "arrayobject.h"
#include "C_arraytest.h"
#include <math.h>
```

```
/* #### Globals ################################### */

/* ==== Set up the methods table ====================== */
static PyMethodDef _C_arraytestMethods[] = {
   // ... skip other defintions
   {"matsq", matsq, METH_VARARGS},
   // ...
   {NULL, NULL}     /* Sentinel - marks the end of this structure */
};

/* ==== Initialize the C_test functions ====================== */
// Module name must be _C_arraytest in compile and linked
void init_C_arraytest()  {
   (void) Py_InitModule("_C_arraytest", _C_arraytestMethods);
   import_array();  // Must be present for NumPy.  Called first after above line.
}

/* ==== Square matrix components function & multiply by int and float ========= */
static PyObject *matsq(PyObject *self, PyObject *args)
{
   PyArrayObject *matin, *matout;
   double **cin, **cout, dfactor;
   int i,j,n,m, dims[2], ifactor;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!id",
      &PyArray_Type, &matin, &ifactor, &dfactor))  return NULL;
   if (NULL == matin)  return NULL;

   /* Check that object input is 'double' type and a matrix
      Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;

   /* Get the dimensions of the input */
   n=dims[0]=matin->dimensions[0];
   m=dims[1]=matin->dimensions[1];

   /* Make a new double matrix of same dims */
   matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);

   /* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pymatrix_to_Carrayptrs(matin);
   cout=pymatrix_to_Carrayptrs(matout);

   /* Do the calculation. */
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         cout[i][j]= ifactor*dfactor*cin[i][j]*cin[i][j];
   }  }

   /* Free memory, close file and return */
   free_Carrayptrs(cin);
   free_Carrayptrs(cout);
   return PyArray_Return(matout);
}
```

Now, lets look at the source code in smaller chunks.

- *Headers*

You must include the following headers with `Python.h` **always** the first header included.

```
#include "Python.h"
#include "arrayobject.h"
```

I also include the header `C_arraytest.h` which contains the prototype of the `matsq` function:

```
static PyObject *matsq(PyObject *self, PyObject *args);
```

The `static` is necessary so objects don't get automatically deleted after the C code is used. The type of the function is `PyObject *` because it will always be returning to a Python calling function so you can (must, actually) return a Python object. The arguments are always the same ,

`PyObject *self` and `PyObject *args`

The first one `self` is never used, but necessary because of how Python passes arguments. The second `args` is a pointer to a Python tuple that contains all of the arguments `(B,i,x)` of the function.

• *Method definitions*

This sets up a table of function names that will be the interface from your Python code to your C extension. The name of the C extension module will be _C_arraytest (note the leading underscore). It is important to get the name right each time it is used because there are strict requirements on using the module name in the code. The name appears first in the method definitions table as the first part of the table name:

```
static PyMethodDef _C_arraytestMethods[] = {
   ...,
   {"matsq", matsq, METH_VARARGS},
   ...,
   {NULL, NULL}     /* Sentinel - marks the end of this structure */
};
```

where I used ellipses (...) to ignore other code not relevant to this function. The `METH_VARARGS` parameter tells the compiler that you will pass the arguments the usual way without keywords as in the example `A=matsq(B,i,x)` above. There are ways to use Python keywords, but I have not tried them out. The table should always end with `{NULL, NULL}` which is just a "marker" to note the end of the table.

• *Initializations*

These functions tell the Python interpreter what to call when the module is loaded. Note the name of the module (`_C_arraytest`) must come directly after the `init` in the name of the initialization structure.

```
void init_C_arraytest()  {
   (void) Py_InitModule("_C_arraytest", _C_arraytestMethods);
   import_array();  // Must be present for NumPy.  Called first after above line.
}
```

The order is important and you must call these two initialization functions first.

• *The* `matsq` *function code*

Now here is the actual function that you will call from Python code. I will split it up and explain each section.

The function name and type:

```
static PyObject *matsq(PyObject *self, PyObject *args)
```

You can see they match the prototype in `C_arraytest.h`.

The local variables:

```
{
   PyArrayObject *matin, *matout;
   double **cin, **cout, dfactor;
   int i,j,n,m, dims[2], ifactor;
```

The PyArrayObjects are structures defined in the NumPy header file and they will be assigned pointers to the actual input and output NumPy arrays (`A` and `B`). The C arrays `cin` and `cout` are C pointers that will point (eventually) to the actual data in the NumPy arrays and allow you to manipulate it. The variable `dfactor` will be the Python float `y, ifactor` will be the Python int `i`, the variables `i,j,n, and m` will be loop variables (`i` and `j`) and matrix dimensions (`n`= number of rows, `m`= number of columns) in `A` and `B`. The array `dims` will be used to access `n` and `m` from the NumPy array. All this happens below. First we have to extract the input variables (`A, i, y`) from the `args` tuple. This is done by the call,

```
   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!id",
      &PyArray_Type, &matin, &ifactor, &dfactor))  return NULL;
```

The `PyArg_ParseTuple` function takes the `args` tuple and using the format string that appears next (`"O!id"`) it assigns each member of the tuple to a C variable. Note you must pass all C variables by reference. This is true even if the C variable is a pointer to a string (see code in `vecfcn1` routine). The format string tells the parsing function what type of variable to use. The common variables for Python all have letter names (e.g. `s` for string, `i` for integer, `d` for (double - the Python float)). You can find a list of these and many more in Guido's tutorial (`http://docs.python.org/ext/ext.html`). For data types that are not in standard Python like the NumPy arrays you use the `O!` notation which tells the parser to look for a type structure (in this case a NumPy structure `PyArray_Type`) to help it convert the tuple member that will be assigned to the local variable (`matin`) pointing to the NumPy array structure. Note these are also passed by reference. The order must be maintained and match the calling interface of the Python function you want. The format string defines the interface and if you do not call the function from Python so the number of arguments match the number in the format string, you will get an error. This is good since it will point to where the problem is.

If this doesn't work we return NULL which will cause a Python exception.

```
   if (NULL == matin)  return NULL;
```

Next we have a check that the input matrix really is a matrix of NumPy type double. This test is also done in the Python wrapper for this C extension. It is better to do it there, but I include the test here to show you that you can do testing in the C extension and you can "reach into" the NumPy structure to pick out it's parameters. The utility function `not_doublematrix` is explained later.

```
   /* Check that object input is 'double' type and a matrix
      Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;
```

Here's an example of reaching into the NumPy structure to get the dimensions of the matrix `matin` and assign them to local variables as mentioned above.

```
   /* Get the dimensions of the input */
   n=dims[0]=matin->dimensions[0];
   m=dims[1]=matin->dimensions[1];
```

Now we use these matrix parameters to generate a new NumPy matrix `matout` (our output) right here in our C extension. `PyArray_FromDims(2,dims,NPY_DOUBLE)` is a utility function provided by NumPy (not me) and its arguments tell NumPy the rank of the NumPy object (`2`), the size of each dimension (`dims`), and the data type (`NPY_DOUBLE`). Other examples of creating different NumPy arrays are in the other C extensions.

```
   /* Make a new double matrix of same dims */
   matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);
```

To actually do our calculations we need C structures to handle our data so we generate two C 2-dimensional arrays (`cin` and `cout`) which will point to the data in `matin` and `matout`, respectively. Note, here memory is allocated since we need to create an array of pointers to C doubles so we can address `cin` and `cout` like usual C matrices with two indices. This memory must be released at the end of this C extension. Memory allocation like this is not always necessary. See the routines for NumPy vector manipulation and treating NumPy matrices like contiguous arrays (as they are in NumPy) in the C extension (the routine `contigmat`).

```
/* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pymatrix_to_Carrayptrs(matin);
   cout=pymatrix_to_Carrayptrs(matout);
```

Finally, we get to the point where we can manipulate the matrices and do our calculations. Here is the part where the original equation operations $B_{ij}= i\ y\ (A_{ij})^2$ are carried out. Note, we are directly manipulating the data in the original NumPy arrays `A` and `B` passed to this extension. So anything you do here to the components of `cin` or `cout` will be done to the original matrices and will appear there when you return to the Python code.

```
   /* Do the calculation. */
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         cout[i][j]= ifactor*dfactor*cin[i][j]*cin[i][j];
   }  }

We are ready to go back to the Python calling routine, but first we release the
memory we allocated for cin and cout.

   /* Free memory, close file and return */
   free_Carrayptrs(cin);
   free_Carrayptrs(cout);

Now we return the result of the calculation.

   return PyArray_Return(matout);
}
```

If you look at the other C extensions you can see that you can also return regular Python variables (like ints) using another Python-provided function `Py_BuildValue("i", 1)` where the string `"i"` tells the function the data type and the second argument (`1` here) is the data value to be returned. If you decide to return nothing, you **must** return the Python keyword None like this:

```
   Py_INCREF(Py_None);
   return Py_None;
```

The `Py_INCREF` function increases the number of references to `None` (remember Python collects allocated memory automatically when there are no more references to the data). You must be careful about this in the C extensions. For more info see that Guido tutorial (`http://docs.python.org/ext/ext.html`).

• *The utility functions*

Here are some quick descriptions of the matrix utility functions. They are pretty much self-explanatory. The vector and integer array utility functions are very similar.

The first utility function is used in one version of a C extension handling matrices (`rowx2_v2`). It does show how one might convert a python object to a NumPy array. It guarantees the memory is contigous which is important for assigning C arrays to the .data part of the structure.

```
PyArrayObject *pymatrix(PyObject *objin)  {
   return (PyArrayObject *) PyArray_ContiguousFromObject(objin,
      NPY_DOUBLE, 2,2);
}
```

The next one creates the C arrays that are used to point to the rows of the NumPy matrices. This allocates arrays of pointers which point into the NumPy data. The NumPy data is contiguous and strides (`m`) are used to access each row. This function calls `ptrvector(n)` which does the actual memory allocation. Remember to deallocate memory after using this one.

```
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin)  {
   double **c, *a;
   int i,n,m;

   n=arrayin->dimensions[0];
   m=arrayin->dimensions[1];
   c=ptrvector(n);
   a=(double *) arrayin->data;  /* pointer to arrayin data as double */
   for ( i=0; i<n; i++)  {
      c[i]=a+i*m;  }
   return c;
}
```

Here is where the memory for the C arrays of pointers is allocated. It's a pretty standard memory allocator for arrays.

```
double **ptrvector(long n)  {
   double **v;
   v=(double **)malloc((size_t) (n*sizeof(double)));
   if (!v)   {
      printf("In **ptrvector. Allocation of memory for double array failed.");
      exit(0);  }
   return v;
}
```

This is the routine to deallocate the memory.

```
void free_Carrayptrs(double **v)  {
   free((char*) v);
}
```

Here is a utility function that checks to make sure the object produced by the parse is a NumPy matrix. You can see how it reaches into the NumPy object structure.

```
int  not_doublematrix(PyArrayObject *mat)  {
   if (mat->descr->type_num != NPY_DOUBLE || mat->nd != 2)  {
      PyErr_SetString(PyExc_ValueError,
         "In not_doublematrix: array must be of type Float and 2 dimensional (n x
m).");
      return 1;  }
   return 0;
}
```

The C Code – other variations

As I mentioned in the introduction the functions are repetitious. All the other functions follow a very similar pattern. They are given a line in the methods structure, they have the same arguments, they parse the arguments, they may check the C structures after the parsing, they set up C variables to manipulate which point to the input data, they do the actual calculation, they deallocate memory (if necessary) and they return something for Python (either `None` or a Python object). I'll just mention some of the differences in the code from the above matrix C extension `matsq`.

`vecfcn1:`

The format string for the parse function specifies that a variable from Python is a string (s).

```
if (!PyArg_ParseTuple(args, "O!O!sd", &PyArray_Type, &vecin,
      &PyArray_Type, &vecout, &str, &dfac))  return NULL;
```

No memory is allocated in the pointer assignments for the local C arrays because they are already vectors.

```
   cin=pyvector_to_Carrayptrs(vecin);
   cout=pyvector_to_Carrayptrs(vecout);
```

The return is an int = 1 if successful. This is returned as a Python int.

```
   return Py_BuildValue("i", 1);
```

`rowx2:`

In this routine we "pass back" the output using the fact that it is passed by reference in the argument tuple list and is changed in place by the manipulations. Compare this to returning an array from the C extension in `matsq`. Either one gets the job done.

`rowx2_v2:`

This routine is similar to the above `rowx2`, but the conversion of the `PyObjects` input (matin and matout) is done outside the `PyArg_ParseTuple` call. `PyArg_ParseTuple` only converts the input into `PyObjects` not `PyArrayObjects` as in `rowx2`. Then the utility function `pymatrix` converts the input `PyObjects to PyArrayObjects` . It seems to be an extra step, but it insures that the `PyObjects` have contigous data, i.e. the data is all sequentially arranged in one memory chunk, not broken up.

`contigmat:`

Here the matrix data is treated like a long vector (just like stacking the rows end to end). This is useful if you have array classes in C++ which store the data as one long vector and then use strides to access it like an array (two-dimensional, three-dimensional, or whatever). Even though `matin` and `matout` are "matrices" we treat them like vectors and use the vector utilities to get our C pointers `cin` and `cout`.

```
/* Change contiguous arrays into C * arrays pointers to PyArrayObject data */
cin=pyvector_to_Carrayptrs(matin);
cout=pyvector_to_Carrayptrs(matout);
```

For other utility functions note that we use different rank, dimensions, and NumPy parameters (e.g. `NPY_LONG`) to tell the routines we are calling what the data types are.

The Make File

The make file is very simple. It is written for Mac OS X 10.4, as BSD Unix.

```
# ---- Link ---------------------------
_C_arraytest.so:  C_arraytest.o  C_arraytest.mak
   gcc -bundle -flat_namespace -undefined suppress -o _C_arraytest.so  C_arraytest.o

# ---- gcc C compile ------------------
C_arraytest.o:  C_arraytest.c C_arraytest.h C_arraytest.mak
   gcc -c C_arraytest.c
-I/Library/Frameworks/Python.framework/Versions/2.4/include/python2.4
-I/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/site-
packages/numpy/core/include/numpy/
```

The compile step is pretty standard. You do need to add paths to the Python headers:
`-I/Library/Frameworks/Python.framework/Versions/2.4/include/python2.4`
and paths to NumPy headers:
`-I/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/site-packages/numpy/core/include/numpy/`

These paths are for a Framework Python 2.4 install on Mac OS X. You need to supply paths to the headers installed on your computer. They may be different. My guess is the gcc flags will be the same for the compile.

The link step produces the actual module (`_C_arraytest.so`) which can be imported to Python code. This is specific to the Mac OS X system. On Linux or Windows you will need something different. I have been searching for generic examples, but I'm not sure what I found would work for most people so I chose not to display the findings there. I cannot judge whether the code is good for those systems.

Note, again the name of the produced shared library **must** match the name in the initialization and methods definition calls in the C extension source code. Hence the leading underline in the name `_C_arraytest.so`.

The Python Wrapper Code

Here as in the C code I will only show detailed description of one wrapper function and its use. There is so much repetition that the other wrappers will be clear if you understand one. I will again use the `matsq` function. This is the code that will first be called when you invoke the `matsq` function in your own Python code after importing the wrapper module (`C_arraytest.py`) which automatically imports and

uses (in a way hidden from the user) the actual C extensions in `_C_arraytest.so` (note the leading underscore which keeps the names separate).

• *imports*

Import the C extensions, NumPy, and the system module (used for the exit statement at the end which is optional).

```
import _C_arraytest
import numpy as NP
import sys
```

• *The definition of the Python matsq function*

Pass a NumPy matrix (`matin`), a Python int (`ifac`), and a Python float (`dfac`).

```
def matsq(matin, ifac, dfac):
```

Check the arguments to make sure they are the right type and dimensions and size. This is much easier and safer on the Python side which is why I do it here even though I showed a way to do some of this in the C extensions.

```
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros((2,2))  # create a NumPy matrix as a test object to check matin
   typetest= type(test)  # get its type
   datatest=test.dtype   # get data type
   if type(matin) != typetest:
      raise 'In matsq, matrix argument is not *NumPy* array'
   if len(NP.shape(matin)) != 2:
      raise 'In matsq, matrix argument is not NumPy *matrix*'
   if matin.dtype != datatest:
      raise 'In matsq, matrix argument is not *Float* NumPy matrix'
   # Make contiguous array if not one
   if not matin.flags.contiguous:
      matin=array(matin)
   if type(ifac) != type(1):
      raise 'In matsq, ifac argument is not an int'
   if type(dfac) != type(1.0):
      raise 'In matsq, dfac argument is not a python float'
```

Finally, call the C extension to do the actual calculation on matin.

```
   # .... Call C extension function
   return _C_arraytest.matsq(matin, ifac, dfac)
```

You can see that the python part is the easiest.

• *Using your C extension*

If the test function mattest2 were in another module (one you were writing), here's how you would use it to call the wrapped matsq function in a script.

```
from C_arraytest import * # NO leading underscore, you are importing the wrapper.
import numpy as NP
import sys

def mattest2():
```

```
   print "\n--- Test matsq ------------------------------"
   print " Each 2nd matrix component should = square of 1st matrix component x ifac x
dfac \n"
   n=4  # Number of columns
   # Make 2 x n matrices
   z=NP.arange(float(n))
   x=NP.array([z,3.0*z])
   jfac=2
   xfac=1.5
   y=matsq(x, jfac, xfac)
   print "x=",x
   print "y=",y

if __name__ == '__main__':

   mattest2()
```

The output looks like this:

```
--- Test matsq ------------------------------
 Each 2nd matrix component should = square of 1st matrix component x ifac x dfac

x= [[ 0.  1.  2.  3.]
 [ 0.  3.  6.  9.]]
y= [[   0.    3.   12.   27.]
 [   0.   27.  108.  243.]]
```

The output of all the test functions is in the file C_arraytest output.

Summary

This is the first draft explaining the C extensions I wrote (with help). If you have comments on the code, mistakes, etc. Please post them on the pythonmac email list. I will see them.

I wrote the C extensions to work with NumPy although they were originally written for Numeric. If you must use Numeric you should test them to see if they are compatible. I suspect names like NPY_DOUBLE, for example, will not be. I strongly suggest you upgrade to the NumPy since it is the future of Numeric in Python. It's worth the effort.

Appendix: Code listings

Carray_test.h

```
/* Header to test of C modules for arrays for Python: C_test.c */

/* ==== Prototypes =================================== */

// .... Python callable Vector functions ..................
static PyObject *vecfcn1(PyObject *self, PyObject *args);
static PyObject *vecsq(PyObject *self, PyObject *args);

/* .... C vector utility functions ..................*/
PyArrayObject *pyvector(PyObject *objin);
```

```
double *pyvector_to_Carrayptrs(PyArrayObject *arrayin);
int  not_doublevector(PyArrayObject *vec);


/* .... Python callable Matrix functions ..................*/
static PyObject *rowx2(PyObject *self, PyObject *args);
static PyObject *rowx2_v2(PyObject *self, PyObject *args);
static PyObject *matsq(PyObject *self, PyObject *args);
static PyObject *contigmat(PyObject *self, PyObject *args);

/* .... C matrix utility functions ..................*/
PyArrayObject *pymatrix(PyObject *objin);
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin);
double **ptrvector(long n);
void free_Carrayptrs(double **v);
int  not_doublematrix(PyArrayObject *mat);

/* .... Python callable integer 2D array functions ..................*/
static PyObject *intfcn1(PyObject *self, PyObject *args);

/* .... C 2D int array utility functions ..................*/
PyArrayObject *pyint2Darray(PyObject *objin);
int **pyint2Darray_to_Carrayptrs(PyArrayObject *arrayin);
int **ptrintvector(long n);
void free_Cint2Darrayptrs(int **v);
int  not_int2Darray(PyArrayObject *mat);
```

Carray_test.c

```
/* A file to test imorting C modules for handling arrays to Python */

#include "Python.h"
#include "arrayobject.h"
#include "C_arraytest.h"
#include <math.h>

/* #### Globals ################################### */

/* ==== Set up the methods table ====================== */
static PyMethodDef _C_arraytestMethods[] = {
   {"vecfcn1", vecfcn1, METH_VARARGS},
   {"vecsq", vecsq, METH_VARARGS},
   {"rowx2", rowx2, METH_VARARGS},
   {"rowx2_v2", rowx2_v2, METH_VARARGS},
   {"matsq", matsq, METH_VARARGS},
   {"contigmat", contigmat, METH_VARARGS},
   {"intfcn1", intfcn1, METH_VARARGS},
   {NULL, NULL}     /* Sentinel - marks the end of this structure */
};

/* ==== Initialize the C_test functions ====================== */
// Module name must be _C_arraytest in compile and linked
void init_C_arraytest()  {
   (void) Py_InitModule("_C_arraytest", _C_arraytestMethods);
   import_array();  // Must be present for NumPy.  Called first after above line.
}

/* #### Vector Extensions ############################## */

/* ==== vector function - manipulate vector in place ======================
```

```
    Multiply the input by 2 x dfac and put in output
    Interface:  vecfcn1(vec1, vec2, str1, d1)
                vec1, vec2 are NumPy vectors,
                str1 is Python string, d1 is Python float (double)
                Returns integer 1 if successful                */
static PyObject *vecfcn1(PyObject *self, PyObject *args)
{
   PyArrayObject *vecin, *vecout;  // The python objects to be extracted from the
args
   double *cin, *cout;             // The C vectors to be created to point to the
                                   //   python vectors, cin and cout point to the row
                                   //   of vecin and vecout, respectively
   int i,j,n;
   const char *str;
   double dfac;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!O!sd", &PyArray_Type, &vecin,
      &PyArray_Type, &vecout, &str, &dfac))  return NULL;
   if (NULL == vecin)  return NULL;
   if (NULL == vecout)  return NULL;

   // Print out input string
   printf("Input string: %s\n", str);

   /* Check that objects are 'double' type and vectors
        Not needed if python wrapper function checks before call to this routine */
   if (not_doublevector(vecin)) return NULL;
   if (not_doublevector(vecout)) return NULL;

   /* Change contiguous arrays into C * arrays   */
   cin=pyvector_to_Carrayptrs(vecin);
   cout=pyvector_to_Carrayptrs(vecout);

   /* Get vector dimension. */
   n=vecin->dimensions[0];

   /* Operate on the vectors  */
   for ( i=0; i<n; i++)  {
         cout[i]=2.0*dfac*cin[i];
   }

   return Py_BuildValue("i", 1);
}

/* ==== Square vector components & multiply by a float ========================
    Returns a NEW  NumPy vector array
    interface:  vecsq(vec1, x1)
                vec1 is NumPy vector, x1 is Python float (double)
                returns a NumPy vector                                    */
static PyObject *vecsq(PyObject *self, PyObject *args)  {
   PyArrayObject *vecin, *vecout;
   double *cin, *cout, dfactor;   // The C vectors to be created to point to the
                                  //   python vectors, cin and cout point to the row
                                  //   of vecin and vecout, respectively
   int i,j,n,m, dims[2];

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!d",
      &PyArray_Type, &vecin, &dfactor))  return NULL;
   if (NULL == vecin)  return NULL;
```

```
   /* Check that object input is 'double' type and a vector
      Not needed if python wrapper function checks before call to this routine */
   if (not_doublevector(vecin)) return NULL;

   /* Get the dimension of the input */
   n=dims[0]=vecin->dimensions[0];

   /* Make a new double vector of same dimension */
   vecout=(PyArrayObject *) PyArray_FromDims(1,dims,NPY_DOUBLE);

   /* Change contiguous arrays into C *arrays   */
   cin=pyvector_to_Carrayptrs(vecin);
   cout=pyvector_to_Carrayptrs(vecout);

   /* Do the calculation. */
   for ( i=0; i<n; i++)  {
         cout[i]= dfactor*cin[i]*cin[i];
   }

   return PyArray_Return(vecout);
}

/* #### Vector Utility functions ######################### */

/* ==== Make a Python Array Obj. from a PyObject, ================
     generates a double vector w/ contiguous memory which may be a new allocation if
     the original was not a double type or contiguous
  !! Must DECREF the object returned from this routine unless it is returned to the
     caller of this routines caller using return PyArray_Return(obj) or
     PyArray_BuildValue with the "N" construct   !!!
*/
PyArrayObject *pyvector(PyObject *objin)  {
   return (PyArrayObject *) PyArray_ContiguousFromObject(objin,
      NPY_DOUBLE, 1,1);
}
/* ==== Create 1D Carray from PyArray ====================
    Assumes PyArray is contiguous in memory.             */
double *pyvector_to_Carrayptrs(PyArrayObject *arrayin)  {
   int i,n;

   n=arrayin->dimensions[0];
   return (double *) arrayin->data;  /* pointer to arrayin data as double */
}
/* ==== Check that PyArrayObject is a double (Float) type and a vector ==============
    return 1 if an error and raise exception */
int  not_doublevector(PyArrayObject *vec)  {
   if (vec->descr->type_num != NPY_DOUBLE || vec->nd != 1)  {
      PyErr_SetString(PyExc_ValueError,
         "In not_doublevector: array must be of type Float and 1 dimensional (n).");
      return 1;  }
   return 0;
}

/* #### Matrix Extensions ############################### */

/* ==== Row x 2 function - manipulate matrix in place ====================
    Multiply the 2nd row of the input by 2 and put in output
    interface:  rowx2(mat1, mat2)
                mat1 and mat2 are NumPy matrices
                Returns integer 1 if successful                        */
static PyObject *rowx2(PyObject *self, PyObject *args)
{
```

```
   PyArrayObject *matin, *matout;  // The python objects to be extracted from the
args
   double **cin, **cout;           // The C matrices to be created to point to the
                                   //   python matrices, cin and cout point to the
rows
                                   //   of matin and matout, respectively
   int i,j,n,m;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!O!", &PyArray_Type, &matin,
      &PyArray_Type, &matout))  return NULL;
   if (NULL == matin)  return NULL;
   if (NULL == matout)  return NULL;

   /* Check that objects are 'double' type and matrices
        Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;
   if (not_doublematrix(matout)) return NULL;

   /* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pymatrix_to_Carrayptrs(matin);
   cout=pymatrix_to_Carrayptrs(matout);

   /* Get matrix dimensions. */
   n=matin->dimensions[0];
   m=matin->dimensions[1];

   /* Operate on the matrices  */
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         if (i==1) cout[i][j]=2.0*cin[i][j];
   }  }

   /* Free memory, close file and return */
   free_Carrayptrs(cin);
   free_Carrayptrs(cout);
   return Py_BuildValue("i", 1);
}
/* ==== Row x 2 function- Version 2. - manipulate matrix in place
======================
    Multiply the 2nd row of the input by 2 and put in output
    interface:  rowx2(mat1, mat2)
                mat1 and mat2 are NumPy matrices
                Returns integer 1 if successful
    Uses the utility function pymatrix to make NumPy C objects from PyObjects
*/
static PyObject *rowx2_v2(PyObject *self, PyObject *args)
{
   PyObject *Pymatin, *Pymatout;   // The python objects to be extracted from the
args
   PyArrayObject *matin, *matout;  // The python array objects to be extracted from
python objects
   double **cin, **cout;           // The C matrices to be created to point to the
                                   //   python matrices, cin and cout point to the
rows
                                   //   of matin and matout, respectively
   int i,j,n,m;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "OO", &Pymatin, &Pymatout))  return NULL;
   if (NULL == Pymatin)  return NULL;
   if (NULL == Pymatout)  return NULL;
```

```
   /* Convert Python Objects to Python Array Objects */
   matin= pymatrix(Pymatin);
   matout= pymatrix(Pymatout);

   /* Check that objects are 'double' type and matrices
        Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;
   if (not_doublematrix(matout)) return NULL;

   /* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pymatrix_to_Carrayptrs(matin);
   cout=pymatrix_to_Carrayptrs(matout);

   /* Get matrix dimensions. */
   n=matin->dimensions[0];
   m=matin->dimensions[1];

   /* Operate on the matrices  */
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         if (i==1) cout[i][j]=2.0*cin[i][j];
   }  }

   /* Free memory, close file and return */
   free_Carrayptrs(cin);
   free_Carrayptrs(cout);
   return Py_BuildValue("i", 1);
}
/* ==== Square matrix components function & multiply by int and float =========
    Returns a NEW NumPy array
    interface:  matsq(mat1, i1, d1)
                mat1 is NumPy matrix, i1 is Python integer, d1 is Python float
(double)
                returns a NumPy matrix                                          */
static PyObject *matsq(PyObject *self, PyObject *args)
{
   PyArrayObject *matin, *matout;
   double **cin, **cout, dfactor;
   int i,j,n,m, dims[2], ifactor;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!id",
      &PyArray_Type, &matin, &ifactor, &dfactor))  return NULL;
   if (NULL == matin)  return NULL;

   /* Check that object input is 'double' type and a matrix
      Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;

   /* Get the dimensions of the input */
   n=dims[0]=matin->dimensions[0];
   m=dims[1]=matin->dimensions[1];

   /* Make a new double matrix of same dims */
   matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);

   /* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pymatrix_to_Carrayptrs(matin);
   cout=pymatrix_to_Carrayptrs(matout);

   /* Do the calculation. */
```

```
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         cout[i][j]= ifactor*dfactor*cin[i][j]*cin[i][j];
   }  }

   /* Free memory, close file and return */
   free_Carrayptrs(cin);
   free_Carrayptrs(cout);
   return PyArray_Return(matout);
}

/* ==== Operate on Matrix components as contiguous memory ========================
  Shows how to access the array data as a contiguous block of memory. Used, for
example,
  in matrix classes implemented as contiquous memory rather than as n arrays of
  pointers to the data "rows"

    Returns a NEW NumPy array
    interface:  contigmat(mat1, x1)
                mat1 is NumPy matrix, x1 is Python float (double)
                returns a NumPy matrix                                          */
static PyObject *contigmat(PyObject *self, PyObject *args)
{
   PyArrayObject *matin, *matout;
   double *cin, *cout, x1;     // Pointers to the contiguous data in the matrices to
                               // be used by C (e.g. passed to a program that uses
                               // matrix classes implemented as contiquous memory
rather
                               // than as n arrays of pointers to the data "rows"
   int i,j,n,m, dims[2], ncomps;  // ncomps=n*m=total number of matrix components in
mat1

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!d",
      &PyArray_Type, &matin, &x1))  return NULL;
   if (NULL == matin)  return NULL;

   /* Check that object input is 'double' type and a matrix
      Not needed if python wrapper function checks before call to this routine */
   if (not_doublematrix(matin)) return NULL;

   /* Get the dimensions of the input */
   n=dims[0]=matin->dimensions[0];
   m=dims[1]=matin->dimensions[1];
   ncomps=n*m;

   /* Make a new double matrix of same dims */
   matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);

   /* Change contiguous arrays into C * arrays pointers to PyArrayObject data */
   cin=pyvector_to_Carrayptrs(matin);
   cout=pyvector_to_Carrayptrs(matout);

   /* Do the calculation. */
   printf("In contigmat, cout (as contiguous memory) =\n");
   for ( i=0; i<ncomps; i++)  {
      cout[i]= cin[i]-x1;
      printf("%e ",cout[i]);
   }
   printf("\n");

   return PyArray_Return(matout);
```

```
}

/* #### Matrix Utility functions ######################### */

/* ==== Make a Python Array Obj. from a PyObject, ================
     generates a double matrix w/ contiguous memory which may be a new allocation if
     the original was not a double type or contiguous
  !! Must DECREF the object returned from this routine unless it is returned to the
     caller of this routines caller using return PyArray_Return(obj) or
     PyArray_BuildValue with the "N" construct   !!!
*/
PyArrayObject *pymatrix(PyObject *objin)  {
   return (PyArrayObject *) PyArray_ContiguousFromObject(objin,
      NPY_DOUBLE, 2,2);
}
/* ==== Create Carray from PyArray =====================
    Assumes PyArray is contiguous in memory.
    Memory is allocated!                                    */
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin)  {
   double **c, *a;
   int i,n,m;

   n=arrayin->dimensions[0];
   m=arrayin->dimensions[1];
   c=ptrvector(n);
   a=(double *) arrayin->data;  /* pointer to arrayin data as double */
   for ( i=0; i<n; i++)  {
      c[i]=a+i*m;  }
   return c;
}
/* ==== Allocate a double *vector (vec of pointers) ====================
    Memory is Allocated!  See void free_Carray(double ** )                  */
double **ptrvector(long n)  {
   double **v;
   v=(double **)malloc((size_t) (n*sizeof(double)));
   if (!v)   {
      printf("In **ptrvector. Allocation of memory for double array failed.");
      exit(0);  }
   return v;
}
/* ==== Free a double *vector (vec of pointers) ========================== */
void free_Carrayptrs(double **v)  {
   free((char*) v);
}
/* ==== Check that PyArrayObject is a double (Float) type and a matrix ==============
    return 1 if an error and raise exception */
int  not_doublematrix(PyArrayObject *mat)  {
   if (mat->descr->type_num != NPY_DOUBLE || mat->nd != 2)  {
      PyErr_SetString(PyExc_ValueError,
         "In not_doublematrix: array must be of type Float and 2 dimensional (n x
m).");
      return 1;  }
   return 0;
}

/* #### Integer 2D Array Extensions ############################### */

/* ==== Integer function - manipulate integer 2D array in place
=====================
    Replace >=0 integer with 1 and < 0 integer with 0 and put in output
    interface:  intfcn1(int1, afloat)
                int1 is a NumPy integer 2D array, afloat is a Python float
```

```
               Returns integer 1 if successful                        */
static PyObject *intfcn1(PyObject *self, PyObject *args)
{
   PyArrayObject *intin, *intout;  // The python objects to be extracted from the
args
   int **cin, **cout;              // The C integer 2D arrays to be created to point
to the
                                   //   python integer 2D arrays, cin and cout point
to the rows
                                   //   of intin and intout, respectively
   int i,j,n,m, dims[2];
   double afloat;

   /* Parse tuples separately since args will differ between C fcns */
   if (!PyArg_ParseTuple(args, "O!d",
      &PyArray_Type, &intin, &afloat))  return NULL;
   if (NULL == intin)  return NULL;

   printf("In intfcn1, the input Python float = %e, a C double\n",afloat);

   /* Check that object input is int type and a 2D array
      Not needed if python wrapper function checks before call to this routine */
   if (not_int2Darray(intin)) return NULL;

   /* Get the dimensions of the input */
   n=dims[0]=intin->dimensions[0];
   m=dims[1]=intin->dimensions[1];

   /* Make a new int array of same dims */
   intout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_LONG);

   /* Change contiguous arrays into C ** arrays (Memory is Allocated!) */
   cin=pyint2Darray_to_Carrayptrs(intin);
   cout=pyint2Darray_to_Carrayptrs(intout);

   /* Do the calculation. */
   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         if (cin[i][j] >= 0)  {
            cout[i][j]= 1;  }
         else  {
            cout[i][j]= 0;  }
   }  }

   printf("In intfcn1, the output array is,\n\n");

   for ( i=0; i<n; i++)  {
      for ( j=0; j<m; j++)  {
         printf("%d ",cout[i][j]);
      }
      printf("\n");
   }
   printf("\n");

   /* Free memory, close file and return */
   free_Cint2Darrayptrs(cin);
   free_Cint2Darrayptrs(cout);
   return PyArray_Return(intout);
}
/* #### Integer Array Utility functions ######################### */

/* ==== Make a Python int Array Obj. from a PyObject, ================
```

```
     generates a 2D integer array w/ contiguous memory which may be a new allocation
if
     the original was not an integer type or contiguous
  !! Must DECREF the object returned from this routine unless it is returned to the
     caller of this routines caller using return PyArray_Return(obj) or
     PyArray_BuildValue with the "N" construct   !!!
*/
PyArrayObject *pyint2Darray(PyObject *objin)  {
   return (PyArrayObject *) PyArray_ContiguousFromObject(objin,
      NPY_LONG, 2,2);
}
/* ==== Create integer 2D Carray from PyArray =====================
    Assumes PyArray is contiguous in memory.
    Memory is allocated!                                    */
int **pyint2Darray_to_Carrayptrs(PyArrayObject *arrayin)  {
   int **c, *a;
   int i,n,m;

   n=arrayin->dimensions[0];
   m=arrayin->dimensions[1];
   c=ptrintvector(n);
   a=(int *) arrayin->data;  /* pointer to arrayin data as int */
   for ( i=0; i<n; i++)  {
      c[i]=a+i*m;  }
   return c;
}
/* ==== Allocate a a *int (vec of pointers) ======================
    Memory is Allocated!  See void free_Carray(int ** )                  */
int **ptrintvector(long n)  {
   int **v;
   v=(int **)malloc((size_t) (n*sizeof(int)));
   if (!v)   {
      printf("In **ptrintvector. Allocation of memory for int array failed.");
      exit(0);  }
   return v;
}
/* ==== Free an int *vector (vec of pointers) ========================== */
void free_Cint2Darrayptrs(int **v)  {
   free((char*) v);
}
/* ==== Check that PyArrayObject is an int (integer) type and a 2D array
==============
    return 1 if an error and raise exception
    Note:  Use NY_LONG for NumPy integer array, not NP_INT      */
int  not_int2Darray(PyArrayObject *mat)  {
   if (mat->descr->type_num != NPY_LONG || mat->nd != 2)  {
      PyErr_SetString(PyExc_ValueError,
         "In not_int2Darray: array must be of type int and 2 dimensional (n x m).");
      return 1;  }
   return 0;
}
```

Make File

```
# ---- Link -------------------------
_C_arraytest.so:  C_arraytest.o  C_arraytest.mak
   gcc -bundle -flat_namespace -undefined suppress -o _C_arraytest.so  C_arraytest.o

# ---- gcc C compile ------------------
C_arraytest.o:  C_arraytest.c C_arraytest.h C_arraytest.mak
```

```
   gcc -c C_arraytest.c -
I/Library/Frameworks/Python.framework/Versions/2.4/include/python2.4 -
I/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/site-
packages/numpy/core/include/numpy/
```

Python Wrappers and Test File

```
#!/usr/local/bin/pythonw

import _C_arraytest
import numpy as NP
import sys

# #### Functions & Classes ##################################################

# ==== Vector functions ========================================

# ---- Test input of 2 vectors and modification of 2nd one ------------
#   Multiply the input by 2 x dfac and put in output
def vecfcn1(vecin, vecout, strin, dfac):
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros(2)
   typetest= type(test)
   datatest=test.dtype
   if type(vecin) != typetest or type(vecout) != typetest:
      raise 'In vecfcn1, arguments are not all *NumPy* arrays'
   if len(NP.shape(vecin)) != 1 or len(NP.shape(vecout)) != 1:
      raise 'In vecfcn1, arguments are not all NumPy *vectors*'
   if vecin.dtype != datatest or vecout.dtype != datatest:
      raise 'In vecfcn1, arguments are not all *Float* NumPy vectors'
   if type(strin) != type("string"):
      raise 'In vecfcn1, strin argument is not a string'
   if type(dfac) != type(1.0):
      raise 'In vecfcn1, dfac argument is not a float'

   # .... Call C extension function
   return _C_arraytest.vecfcn1(vecin, vecout, strin, dfac)

# ---- Test input 1 vector and creation of 2nd one ------------
#   Each 2nd vector component should = square of 1st vector component x xfac
def vecsq(vecin, xfac):
   # .... Check arguments, double NumPy vector?
   test=NP.zeros(2)
   typetest= type(test)
   datatest=test.dtype
   if type(vecin) != typetest:
      raise 'In vecsq, vector argument is not *NumPy* array'
   if len(NP.shape(vecin)) != 1:
      raise 'In vecsq, vector argument is not NumPy *vector*'
   if vecin.dtype != datatest:
      raise 'In vecsq, vector argument is not *Float* NumPy vector'
   if type(xfac) != type(1.0):
      raise 'In vecsq, xfac argument is not a python float'

   # .... Call C extension function
   return _C_arraytest.vecsq(vecin, xfac)

#==== Vector Tests ============================

# ---- Test rowx2 ----------------------------
```

```
#   Manipulation in place
#   Multiply the input by 2 x dfac and put in output
def vectest1():
   print "\n--- Test vecfcn1 ------------------------------"
   print " Multiply the input by 2 x dfac and put in output \n"
   n=4  # Number of columns
   # Make 2 vectors
   x=NP.arange(float(n))
   y=NP.array(x)  # generate a copy of x (will be changed in vecfcn1 call)
   st="I'm in a C extension."
   df=2.0
   vecfcn1(x,y,st,df)
   print "x=",x
   print "y=",y

# ---- Test vecsqr ------------------------------
#   Create new array from input and return new NumPy array
#   Each 2nd vector component should = square of 1st vector component x xfac
def vectest2():
   print "\n--- Test vecsq ------------------------------"
   print " Each 2nd vector component should = square of 1st vector component x xfac
\n"
   n=7  # Number of columns
   x=NP.arange(float(n))
   xfac= -2.5
   y=vecsq(x, xfac)
   print "x=",x
   print "y=",y

# ==== Matrix functions ============================================

# ---- Test input of 2 matrices and modification of 2nd one ------------
#   2nd row of matrix y should be 2 x 2nd row of matrix x
#   (PyArrayObject construction directly from PyArg_ParseTuple)
def rowx2(matin, matout):
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros((2,2))
   typetest= type(test)
   datatest=test.dtype
   if type(matin) != typetest or type(matout) != typetest:
      raise 'In rowx2, arguments are not all *NumPy* arrays'
   if len(NP.shape(matin)) != 2 or len(NP.shape(matout)) != 2:
      raise 'In rowx2, arguments are not all NumPy *matrices*'
   if matin.dtype != datatest or matout.dtype != datatest:
      raise 'In rowx2, arguments are not all *Float* NumPy matrices'

   # .... Call C extension function
   return _C_arraytest.rowx2(matin, matout)

# ---- Test input of 2 matrices and modification of 2nd one ------------
#   2nd row of matrix y should be 2 x 2nd row of matrix x
#   (PyArrayObject construction different from rowx2)
def rowx2_v2(matin, matout):
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros((2,2))
   typetest= type(test)
   datatest=test.dtype
   if type(matin) != typetest or type(matout) != typetest:
      raise 'In rowx2, arguments are not all *NumPy* arrays'
   if len(NP.shape(matin)) != 2 or len(NP.shape(matout)) != 2:
      raise 'In rowx2, arguments are not all NumPy *matrices*'
   if matin.dtype != datatest or matout.dtype != datatest:
```

```
      raise 'In rowx2, arguments are not all *Float* NumPy matrices'

   # .... Call C extension function
   return _C_arraytest.rowx2_v2(matin, matout)

# ---- Test input 1 matrices and creation of 2nd one ------------
#   Each 2nd matrix component should = square of 1st matrix component x ifac x dfac
def matsq(matin, ifac, dfac):
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros((2,2))
   typetest= type(test)
   datatest=test.dtype
   if type(matin) != typetest:
      raise 'In matsq, matrix argument is not *NumPy* array'
   if len(NP.shape(matin)) != 2:
      raise 'In matsq, matrix argument is not NumPy *matrix*'
   if matin.dtype != datatest:
      raise 'In matsq, matrix argument is not *Float* NumPy matrix'
   if type(ifac) != type(1):
      raise 'In matsq, ifac argument is not an int'
   if type(dfac) != type(1.0):
      raise 'In matsq, dfac argument is not a python float'

   # .... Call C extension function
   return _C_arraytest.matsq(matin, ifac, dfac)

# ---- Test contiguous memory treatment of matrices ---------------
#   Each 2nd matrix component should = 1st matrix component - x1
#   In the extension function _C_arraytest.contigmat the matrix data
#   is handled as contiguous memory *not* as arrays of pointers to data rows
#   like in typical C matrices.
def contigmat(matin, x1):
   # .... Check arguments, double NumPy matrices?
   test=NP.zeros((2,2))
   typetest= type(test)
   datatest=test.dtype
   if type(matin) != typetest:
      raise 'In contigmat, matrix argument is not *NumPy* array'
   if len(NP.shape(matin)) != 2:
      raise 'In contigmat, matrix argument is not NumPy *matrix*'
   if matin.dtype != datatest:
      raise 'In contigmat, matrix argument is not *Float* NumPy matrix'
   if type(x1) != type(1.0):
      raise 'In contigmat,  x1 argument is not a python float'

   # .... Call C extension function
   return _C_arraytest.contigmat(matin, x1)

#==== Matrix Tests ==============================

# ---- Test rowx2 & rowx2_v2 -----------------------------
#   Manipulation in place
#   2nd row of matrix y should be 2 x 2nd row of matrix x
#  rowx2_v2 converts PyObjects to PyArrayObjects differently than rowx2.
#
def mattest1():
   print "\n--- Test rowx2 -----------------------------"
   print "  (PyArrayObject construction directly from PyArg_ParseTuple)"
   print " 2nd row of matrix y should be 2 x 2nd row of matrix x \n"
   n=4  # Number of columns
   # Make 2 x n matrices
   z=NP.arange(float(n))
```

```
   x=NP.array([z,z])
   y=NP.array(x)  # generate a copy of x (will be changed in rowx2 call)
   rowx2(x,y)
   print "x=",x
   print "y=",y
   print "\n--- Test rowx2_v2  -----------------------------"
   print "  (PyArrayObject construction different from rowx2)"
   print " 2nd row of matrix y should be 2 x 2nd row of matrix x \n"
   w=NP.array(x)  # generate a copy of x (will be changed in rowx2 call)
   rowx2(x,w)
   print "x=",x
   print "w=",w


# ---- Test matsqr -----------------------------
#   Create new array from input and return new NumPy array
#   Each 2nd matrix component should = square of 1st matrix component x ifac x dfac
def mattest2():
   print "\n--- Test matsq -----------------------------"
   print " Each 2nd matrix component should = square of 1st matrix component x ifac x
dfac \n"
   n=4  # Number of columns
   # Make 2 x n matrices
   z=NP.arange(float(n))
   x=NP.array([z,3.0*z])
   jfac=2
   xfac=1.5
   y=matsq(x, jfac, xfac)
   print "x=",x
   print "y=",y

# ---- Test contigmat -----------------------------
#   Create new array from input and return new NumPy array
#   Each 2nd matrix component should = square of 1st matrix component x ifac x dfac
def mattest3():
   print "\n--- Test contigmat -----------------------------"
   print " Each 2nd matrix component should = square of 1st matrix component x ifac x
dfac \n"
   n=4  # Number of columns
   # Make 2 x n matrices
   z=NP.arange(float(n))
   x=NP.array([z,3.0*z])
   x1=1.5
   y=contigmat(x, x1)
   print "x=",x
   print "y=",y

# ==== Integer 2D array functions ============================================

# ---- Test input 1 integer 2D array and creation of 2nd one ------------
#   Each 2nd matrix component should = 1 if intin >= 0 or 0 otherwise
def intfcn1(intin, afloat):
   # .... Check arguments, integer NumPy 2D array?
   test=NP.zeros((2,2),dtype='i')
   typetest= type(test)
   datatest=test.dtype
   if type(intin) != typetest:
      raise 'In intfcn1, argument is not *NumPy* array'
   if len(NP.shape(intin)) != 2:
      raise 'In intfcn1, argument is not NumPy *integer 2D array*'
   if intin.dtype != datatest:
      raise 'In intfcn1, input argument is not *integer* NumPy 2D array'
```

```
   if type(afloat) != type(1.0):
      raise 'In intfcn1, afloat argument is not a python float'

   # .... Call C extension function
   return _C_arraytest.intfcn1(intin, afloat)

#==== Integer 2D array Tests ============================

# ---- Test matsqr ----------------------------
#   Create new array from input and return new NumPy array
#   Each 2nd matrix component should = square of 1st matrix component x ifac x dfac
def intarrtest1():
   print "\n--- Test intarrtest1 ----------------------------"
   print " Each 2nd matrix component should = 1 if intin >= 0 or 0 otherwise \n"
   n=5  # Number of columns
   # Make 2 x n 2D integer arrays
   z=NP.arange(int(n))
   x=NP.array([z,3*z])
   x=x-3
   afloat= -22.78
   y=intfcn1(x, afloat)
   print "x=",x
   print "y=",y



# #### Run code ########################################################
if __name__ == '__main__':

   vectest1()
   vectest2()
   mattest1()
   mattest2()
   mattest3()
   intarrtest1()

# ###### STOP HERE ###################################################
   sys.exit()
```