numpy.i: a SWIG Interface File for NumPy

Author:	Bill Spotz
Institution:	Sandia National Laboratories
Date:	13 September, 2007

Contents

Introduction

The Simple Wrapper and Interface Generator (or SWIG) is a powerful tool for generating wrapper code for interfacing to a wide variety of scripting languages. SWIG can parse header files, and using only the code prototypes, create an interface to the target language. But SWIG is not omnipotent. For example, it cannot know from the prototype:

```
double rms(double* seq, int n);
```

what exactly `seq` is. Is it a single value to be altered in-place? Is it an array, and if so what is its length? Is it input-only? Output-only? Input-output? SWIG cannot determine these details, and does not attempt to do so.

Making an educated guess, humans can conclude that this is probably a routine that takes an input-only array of length `n` of `double` values called `seq` and returns the root mean square. The default behavior of SWIG, however, will be to create a wrapper function that compiles, but is nearly impossible to use from the scripting language in the way the C routine was intended.

For python, the preferred way of handling contiguous (or technically, *strided*) blocks of homogeneous data is with the module NumPy, which provides full object-oriented access to arrays of data. Therefore, the most logical python interface for the `rms` function would be (including doc string):

```
def rms(seq):
    """
    rms: return the root mean square of a sequence
    rms(numpy.ndarray) -> double
    rms(list) -> double
    rms(tuple) -> double
    """
```

where `seq` would be a NumPy array of `double` values, and its length `n` would be extracted from `seq` internally before being passed to the C routine. Even better, since NumPy supports construction of arrays from arbitrary python sequences, `seq` itself could be a nearly arbitrary sequence (so long as each element can be converted to a `double`) and the wrapper code would internally convert it to a NumPy array before extracting its data and length.

SWIG allows these types of conversions to be defined via a mechanism called typemaps. This document provides information on how to use `numpy.i`, a SWIG interface file that defines a series of typemaps intended to make the type of array-related conversions described above relatively simple to implement. For example, suppose that the `rms` function prototype defined above was in a header file named `rms.h`. To obtain the python interface discussed above, your SWIG interface file would need the following:

```
%{
#define SWIG_FILE_WITH_INIT
#include "rms.h"
%}

%include "numpy.i"

%init %{
import_array();
%}

%apply (double* IN_ARRAY1, int DIM1) {(double* seq, int n)};
%include "rms.h"
```

Typemaps are keyed off a list of one or more function arguments, either by type or by type and name. We will refer to such lists as *signatures*. One of the many typemaps defined by `numpy.i` is used above and has the signature `(double* IN_ARRAY1, int DIM1)`. The argument names are intended to suggest that the `double*` argument is an input array of one dimension and that the `int` represents that dimension. This is precisely the pattern in the `rms` prototype.

Most likely, no actual prototypes to be wrapped will have the argument names `IN_ARRAY1` and `DIM1`. We use the `%apply` directive to apply the typemap for one-dimensional input arrays of type `double` to the actual prototype used by `rms`. Using `numpy.i` effectively, therefore, requires knowing what typemaps are available and what they do.

A SWIG interface file that includes the SWIG directives given above will produce wrapper code that looks something like:

```
 1 PyObject *_wrap_rms(PyObject *args) {
 2   PyObject *resultobj = 0;
 3   double *arg1 = (double *) 0 ;
 4   int arg2 ;
 5   double result;
 6   PyArrayObject *array1 = NULL ;
 7   int is_new_object1 = 0 ;
 8   PyObject * obj0 = 0 ;
 9
10   if (!PyArg_ParseTuple(args,(char *)"O:rms",&obj0)) SWIG_fail;
11   {
12     array1 = obj_to_array_contiguous_allow_conversion(
13                  obj0, NPY_DOUBLE, &is_new_object1);
14     npy_intp size[1] = {
15       -1
16     };
17     if (!array1 || !require_dimensions(array1, 1) ||
18         !require_size(array1, size, 1)) SWIG_fail;
19     arg1 = (double*) array1->data;
20     arg2 = (int) array1->dimensions[0];
21   }
22   result = (double)rms(arg1,arg2);
23   resultobj = SWIG_From_double((double)(result));
24   {
25     if (is_new_object1 && array1) Py_DECREF(array1);
26   }
27   return resultobj;
28 fail:
29   {
30     if (is_new_object1 && array1) Py_DECREF(array1);
31   }
32   return NULL;
33 }
```

The typemaps from `numpy.i` are responsible for the following lines of code: 12--20, 25 and 30. Line 10 parses the input to the `rms` function. From the format string `"O:rms"`, we can see that the argument list is expected to be a single python object (specified by the `O` before the colon) and whose pointer is stored in `obj0`. A number of functions, supplied by `numpy.i`, are called to make and check the (possible) conversion from a generic python object to a NumPy array. These functions are explained in the section Helper Functions, but hopefully their names are self-explanatory. At line 12 we use `obj0` to construct a NumPy array. At line 17, we check the validity of the result: that it is non-null and that it has a single dimension of arbitrary length. Once these states are verified, we extract the data buffer and length in lines 19 and 20 so that we can call the underlying C function at line 22. Line 25 performs memory management for the case where we have created a new array that is no longer needed.

This code has a significant amount of error handling. Note the `SWIG_fail` is a macro for `goto fail`, refering to the label at line 28. If the user provides the wrong number of arguments, this will be caught at line 10. If construction of the NumPy array fails or produces an array with the wrong number of dimensions, these errors are caught at line 17. And finally, if an error is detected, memory is still managed correctly at line 30.

Note that if the C function signature was in a different order:

```
double rms(int n, double* seq);
```

that SWIG would not match the typemap signature given above with the argument list for `rms`. Fortunately, `numpy.i` has a set of typemaps with the data pointer given last:

```
%apply (int DIM1, double* IN_ARRAY1) {(int n, double* seq)};
```

This simply has the effect of switching the definitions of `arg1` and `arg2` in lines 3 and 4 of the generated code above, and their assignments in lines 19 and 20.

Using numpy.i

The `numpy.i` file is currently located in the `numpy/docs/swig` sub-directory under the `numpy` installation directory. Typically, you will want to copy it to the directory where you are developing your wrappers. If it is ever adopted by SWIG developers, then it will be installed in a standard place where SWIG can find it.

A simple module that only uses a single SWIG interface file should include the following:

```
%{
#define SWIG_FILE_WITH_INIT
%}
%include "numpy.i"
%init %{
import_array();
%}
```

Within a compiled python module, `import_array()` should only get called once. This could be in a C/C++ file that you have written and is linked to the module. If this is the case, then none of your interface files should `#define SWIG_FILE_WITH_INIT` or call `import_array()`. Or, this initialization call could be in a wrapper file generated by SWIG from an interface file that has the `%init` block as above. If this is the case, and you have more than one SWIG interface file, then only one interface file should `#define SWIG_FILE_WITH_INIT` and call `import_array()`.

Available Typemaps

The typemap directives provided by `numpy.i` for arrays of different data types, say `double` and `int`, and dimensions of different types, say `int` or `long`, are identical to one another except for the C and NumPy type specifications. The typemaps are therefore implemented (typically behind the scenes) via a macro:

```
%numpy_typemaps(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)
```

that can be invoked for appropriate `(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)` triplets. For example:

```
%numpy_typemaps(double, NPY_DOUBLE, int)
%numpy_typemaps(int,    NPY_INT   , int)
```

The `numpy.i` interface file uses the `%numpy_typemaps` macro to implement typemaps for the following C data types and `int` dimension types:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`

- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- `float`
- `double`

In the following descriptions, we reference a generic `DATA_TYPE`, which could be any of the C data types listed above.

Input Arrays

Input arrays are defined as arrays of data that are passed into a routine but are not altered in-place or returned to the user. The python input array is therefore allowed to be almost any python sequence (such as a list) that can be converted to the requested type of array. The input array signatures are

- `( DATA_TYPE IN_ARRAY1[ANY] )`
- `( DATA_TYPE* IN_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* IN_ARRAY1 )`
- `( DATA_TYPE IN_ARRAY2[ANY][ANY] )`
- `( DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2 )`
- `( DATA_TYPE IN_ARRAY3[ANY][ANY][ANY] )`
- `( DATA_TYPE* IN_ARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_ARRAY3 )`

The first signature listed, `( DATA_TYPE IN_ARRAY[ANY] )` is for one-dimensional arrays with hard-coded dimensions. Likewise, `( DATA_TYPE IN_ARRAY2[ANY][ANY] )` is for two-dimensional arrays with hard-coded dimensions, and similarly for three-dimensional.

In-Place Arrays

In-place arrays are defined as arrays that are modified in-place. The input values may or may not be used, but the values at the time the function returns are significant. The provided python argument must therefore be a NumPy array of the required type. The in-place signatures are

- `( DATA_TYPE INPLACE_ARRAY1[ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* INPLACE_ARRAY1 )`
- `( DATA_TYPE INPLACE_ARRAY2[ANY][ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* INPLACE_ARRAY2 )`
- `( DATA_TYPE INPLACE_ARRAY3[ANY][ANY][ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_ARRAY3 )`

These typemaps now check to make sure that the `INPLACE_ARRAY` arguments use native byte ordering. If not, an exception is raised.

Argout Arrays

Argout arrays are arrays that appear in the input arguments in C, but are in fact output arrays. This pattern occurs often when there is more than one output variable and the single return argument is therefore not sufficient. In python, the convential way to return multiple arguments is to pack them into a sequence (tuple, list, etc.) and return the sequence. This is what the argout typemaps do. If a wrapped function that uses these argout typemaps has more than one return argument, they are packed into a tuple or list, depending on the version of python. The python user does not pass these arrays in, they simply get returned. The argout signatures are

- `( DATA_TYPE ARGOUT_ARRAY1[ANY] )`
- `( DATA_TYPE* ARGOUT_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* ARGOUT_ARRAY1 )`
- `( DATA_TYPE ARGOUT_ARRAY2[ANY][ANY] )`
- `( DATA_TYPE ARGOUT_ARRAY3[ANY][ANY][ANY] )`

These are typically used in situations where in C/C++, you would allocate a(n) array(s) on the heap, and call the function to fill the array(s) values. In python, the arrays are allocated for you and returned as new array objects.

Note that we support `DATA_TYPE*` argout typemaps in 1D, but not 2D or 3D. This because of a quirk with the SWIG typemap syntax and cannot be avoided. Note that for these types of 1D typemaps, the python function will take a single argument representing `DIM1`.

Output Arrays

The `numpy.i` interface file does not support typemaps for output arrays, for several reasons. First, C/C++ return arguments are limited to a single value. This prevents obtaining dimension information in a general way. Second, arrays with hard-coded lengths are not permitted as return arguments. In other words:

```
double[3] newVector(double x, double y, double z);
```

is not legal C/C++ syntax. Therefore, we cannot provide typemaps of the form:

```
%typemap(out) (TYPE[ANY]);
```

If you run into a situation where a function or method is returning a pointer to an array, your best bet is to write your own version of the function to be wrapped, either with `%extend` for the case of class methods or `%ignore` and `%rename` for the case of functions.

Other Common Types: bool

Note that C++ type `bool` is not supported in the list in the Available Typemaps section. NumPy bools are a single byte, while the C++ `bool` is four bytes (at least on my system). Therefore:

```
%numpy_typemaps(bool, NPY_BOOL, int)
```

will result in typemaps that will produce code that reference improper data lengths. You can implement the following macro expansion:

```
%numpy_typemaps(bool, NPY_UINT, int)
```

to fix the data length problem, and Input Arrays will work fine, but In-Place Arrays might fail type-checking.

Other Common Types: complex

Typemap conversions for complex floating-point types is also not supported automatically. This is because python and NumPy are written in C, which does not have native complex types. Both python and NumPy implement their own (essentially equivalent) `struct` definitions for complex variables:

```
/* Python */
typedef struct {double real; double imag;} Py_complex;

/* NumPy */
typedef struct {float  real, imag;} npy_cfloat;
typedef struct {double real, imag;} npy_cdouble;
```

We could have implemented:

```
%numpy_typemaps(Py_complex , NPY_CDOUBLE, int)
%numpy_typemaps(npy_cfloat , NPY_CFLOAT , int)
%numpy_typemaps(npy_cdouble, NPY_CDOUBLE, int)
```

which would have provided automatic type conversions for arrays of type `Py_complex`, `npy_cfloat` and `npy_cdouble`. However, it seemed unlikely that there would be any independent (non-python, non-NumPy) application code that people would be using SWIG to generate a python interface to, that also used these definitions for complex types. More likely, these application codes will define their own complex types, or in the case of C++, use `std::complex`. Assuming these data structures are compatible with python and NumPy complex types, `%numpy_typemap` expansions as above (with the user's complex type substituted for the first argument) should work.

Helper Functions

The `numpy.i` file containes several macros and routines that it uses internally to build its typemaps. However, these functions may be useful elsewhere in your interface file.

Macros

is_array(a) Evaluates as true if `a` is non-`NULL` and can be cast to a `PyArrayObject*`.

array_type(a) Evaluates to the integer data type code of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_numdims(a) Evaluates to the integer number of dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_dimensions(a) Evaluates to an array of type `npy_intp` and length `array_numdims(a)`, giving the lengths of all of the dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_size(a,i) Evaluates to the `i`-th dimension size of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_data(a) Evaluates to a pointer of type `void*` that points to the data buffer of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_is_contiguous(a) Evaluates as true if `a` is a contiguous array. Equivalent to `(PyArray_ISCONTIGUOUS(a))`.

array_is_native(a) Evaluates as true if the data buffer of `a` uses native byte order. Equivalent to `(PyArray_ISNOTSWAPPED(a))`.

Routines

pytype_string()

Return type: `char*`

Arguments:

- `PyObject* py_obj`, a general python object.

Return a string describing the type of `py_obj`.

typecode_string()

Return type: `char*`

Arguments:

- `int typecode`, a NumPy integer typecode.

Return a string describing the type corresponding to the NumPy `typecode`.

type_match()

Return type: `int`

Arguments:

- `int actual_type`, the NumPy typecode of a NumPy array.
- `int desired_type`, the desired NumPy typecode.

Make sure that `actual_type` is compatible with `desired_type`. For example, this allows character and byte types, or int and long types, to match. This is now equivalent to `PyArray_EquivTypenums()`.

obj_to_array_no_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general python object.
- `int typecode`, the desired NumPy typecode.

Cast `input` to a `PyArrayObject*` if legal, and ensure that it is of type `typecode`. If `input` cannot be cast, or the `typecode` is wrong, set a python error and return `NULL`.

obj_to_array_allow_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a NumPy array with the given `typecode`. On success, return a valid `PyArrayObject*` with the correct type. On failure, the python error string will be set and the routine returns `NULL`.

make_contiguous()

Return type: `PyArrayObject*`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.
- `int min_dims`, minimum allowable dimensions.
- `int max_dims`, maximum allowable dimensions.

Check to see if `ary` is contiguous. If so, return the input pointer and flag it as not a new object. If it is not contiguous, create a new `PyArrayObject*` using the original data, flag it as a new object and return the pointer.

obj_to_array_contiguous_allow_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a contiguous `PyArrayObject*` of the specified type. If the input object is not a contiguous `PyArrayObject*`, a new one will be created and the new object flag will be set.

require_contiguous()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

Test whether `ary` is contiguous. If so, return 1. Otherwise, set a python error and return 0.

require_native()

Return type: `int`

Arguments:

- `PyArray_Object* ary`, a NumPy array.

Require that `ary` is not byte-swapped. If the array is not byte-swapped, return 1. Otherwise, set a python error and return 0.

require_dimensions()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int exact_dimensions`, the desired number of dimensions.

Require `ary` to have a specified number of dimensions. If the array has the specified number of dimensions, return 1. Otherwise, set a python error and return 0.

require_dimensions_n()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int* exact_dimensions`, an array of integers representing acceptable numbers of dimensions.
- `int n`, the length of `exact_dimensions`.

Require `ary` to have one of a list of specified number of dimensions. If the array has one of the specified number of dimensions, return 1. Otherwise, set the python error string and return 0.

require_size()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `npy_int* size`, an array representing the desired lengths of each dimension.
- `int n`, the length of `size`.

Require `ary` to have a specified shape. If the array has the specified shape, return 1. Otherwise, set the python error string and return 0.

Beyond the Provided Typemaps

There are many C or C++ array/NumPy array situations not covered by a simple `%include "numpy.i"` and subsequent `%apply` directives.

A Common Example

Consider a reasonable prototype for a dot product function:

```
double dot(int len, double* vec1, double* vec2);
```

The python interface that we want is:

```
def dot(vec1, vec2):
    """
    dot(PyObject,PyObject) -> double
    """
```

The problem here is that there is one dimension argument and two array arguments, and our typemaps are set up for dimensions that apply to a single array (in fact, SWIG does not provide a mechanism for associating `len` with `vec2` that takes two python input arguments). The recommended solution is the following:

```
%apply (int DIM1, double* IN_ARRAY1) {(int len1, double* vec1),
                                      (int len2, double* vec2)}
%rename (dot) my_dot;
%exception my_dot {
    $action
    if (PyErr_Occurred()) SWIG_fail;
}
%inline %{
double my_dot(int len1, double* vec1, int len2, double* vec2) {
    if (len1 != len2) {
        PyErr_Format(PyExc_ValueError,
                     "Arrays of lengths (%d,%d) given",
                     len1, len2);
        return 0.0;
    }
    return dot(len1, vec1, vec2);
}
%}
```

If the header file that contains the prototype for `double dot()` also contains other prototypes that you want to wrap, so that you need to `%include` this header file, then you will also need a `%ignore dot;` directive, placed after the `%rename` and before the `%include` directives. Or, if the function in question is a class method, you will want to use `%extend` rather than `%inline` in addition to `%ignore`.

A note on error handling: Note that `my_dot` returns a `double` but that it can also raise a python error. The resulting wrapper function will return a python float representation of 0.0 when the vector lengths do not match. Since this is not `NULL`, the python interpreter will not know to check for an error. For this reason, we add the `%exception` directive above for `my_dot` to get the behavior we want (note that `$action` is a macro that gets expanded to a valid call to `my_dot`). In general, you will probably want to write a SWIG macro to perform this task.

Other Situations

There are other wrapping situations in which `numpy.i` may be helpful when you encounter them.

- In some situations, it is possible that you could use the `%numpy_templates` macro to implement typemaps for your own types. See the Other Common Types: bool or Other Common Types: complex sections for examples. Another situation is if your dimensions are of a type other than `int` (say `long` for example):

  ```
  %numpy_typemaps(double, NPY_DOUBLE, long)
  ```

- You can use the code in `numpy.i` to write your own typemaps. For example, if you had a four-dimensional array as a function argument, you could cut-and-paste the appropriate three-dimensional typemaps into your interface file. The modifications for the fourth dimension would be trivial.
- Sometimes, the best approach is to use the `%extend` directive to define new methods for your classes (or overload existing ones) that take a `PyObject*` (that either is or can be converted to a `PyArrayObject*`) instead of a pointer to a buffer. In this case, the helper routines in `numpy.i` can be very useful.
- Writing typemaps can be a bit nonintuitive. If you have specific questions about writing SWIG typemaps for NumPy, the developers of `numpy.i` do monitor the Numpy-discussion and Swig-user mail lists.

A Final Note

When you use the `%apply` directive, as is usually necessary to use `numpy.i`, it will remain in effect until you tell SWIG that it shouldn't be. If the arguments to the functions or methods that you are wrapping have common names, such as `length` or `vector`, these typemaps may get applied in situations you do not expect or want. Therefore, it is always a good idea to add a `%clear` directive after you are done with a specific typemap:

```
%apply (double* IN_ARRAY1, int DIM1) {(double* vector, int length)}
%include "my_header.h"
%clear (double* vector, int length);
```

In general, you should target these typemap signatures specifically where you want them, and then clear them after you are done.

Summary

Out of the box, `numpy.i` provides typemaps that support conversion between NumPy arrays and C arrays:

- That can be one of 12 different scalar types: `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` and `double`.
- That support 23 different argument signatures for each data type, including:
  - One-dimensional, two-dimensional and three-dimensional arrays.
  - Input-only, in-place, and argout behavior.
  - Hard-coded dimensions, data-buffer-then-dimensions specification, and dimensions-then-data-buffer specification.

The `numpy.i` interface file also provides additional tools for wrapper developers, including:

- A SWIG macro (`%numpy_typemaps`) with three arguments for implementing the 23 argument signatures for the user's choice of (1) C data type, (2) NumPy data type (assuming they match), and (3) dimension type.
- Seven C macros and eleven C functions that can be used to write specialized typemaps, extensions, or inlined functions that handle cases not covered by the provided typemaps.

Acknowledgements

Many people have worked to glue SWIG and NumPy together (as well as SWIG and the predecessors of NumPy, Numeric and numarray). The effort to standardize this work into `numpy.i` began at the 2005 SciPy Conference with a conversation between Fernando Perez and myself. Fernando collected helper functions and typemaps from Michael Hunter, Anna Omelchenko and Michael Sanner. Sebastian Hasse has also provided additional error checking and use cases. The work of these contributors has made this end result possible.

Generated on: 2007-09-13 17:08 UTC. Generated by Docutils from reStructuredText source.