numpy.i: a SWIG Interface File for NumPy

Author: Bill Spotz
Institution: Sandia National Laboratories
Date: 18 March, 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:

```
def rms(seq):
```

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.

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)};
```

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, int DIM1)`
- `(DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2)`
- `(int DIM1, DATA_TYPE* IN_ARRAY1)`
- `(int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2)`

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, int DIM1)`
- `(DATA_TYPE* INPLACE_ARRAY2, int DIM1, int DIM2)`
- `(int DIM1, DATA_TYPE* INPLACE_ARRAY1)`
- `(int DIM1, int DIM2, DATA_TYPE* INPLACE_ARRAY2)`

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 tuple and return the tuple. This is what the argout typemaps do. If a wrapped function that uses these argout typemaps has more than one return argument, they are so packed. 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_ARRAY2[ANY][ANY])`

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_DOUBLE , 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_dimensions(a) Evaluates to the integer number of 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_is_contiguous(a) Evaluates as true if `a` is a contiguous array. Equivalent to `(PyArray_ISCONTIGUOUS(a))`.

Routines

char* pytype_string(PyObject* py_obj) Given a `PyObject*`, return a string describing its type.

char* typecode_string(int typecode) Given a NumPy integer typecode, return a string describing the type.

int type_match(int actual_type, int desired_type) Make sure input has correct NumPy type. Allow character and byte to match. Also allow int and long to match.

PyArrayObject* obj_to_array_no_conversion(PyObject* input, int typecode) Given a `PyObject*`, cast it to a `PyArrayObject*` if legal. If not, set the python error string appropriately and return `NULL`.

PyArrayObject* obj_to_array_allow_conversion(PyObject* input, int typecode, int* is_new_object) Convert the given `PyObject*` 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`.

PyArrayObject* make_contiguous(PyArrayObject* ary, int* is_new_object, int min_dims, int max_d Given a `PyArrayObject*`, check to see if it 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.

PyArrayObject* obj_to_array_contiguous_allow_conversion(PyObject* input, int typecode, int* is_n Convert a given `PyObject*` 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.

int require_contiguous(PyArrayObject* ary) Test whether a `PyArrayObject*` is contiguous. If array is contiguous, return 1. Otherwise, set the python error string and return 0.

int require_dimensions(PyArrayObject* ary, int exact_dimensions) Require the given `PyArrayObject*` to have a specified number of dimensions. If the array has the specified number of dimensions, return 1. Otherwise, set the python error string and return 0.

int require_dimensions_n(PyArrayObject* ary, int* exact_dimensions, int n) Require the given `PyArrayObject*` 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.

int require_size(PyArrayObject* ary, int* size, int n) Require the given `PyArrayObject*` to have a specified shape. If the array has the specified shape, return 1. Otherwise, set the python error string and return 0.

static PyArrayObject *contiguous_typed_array(PyObject *obj, int typecode, int expectnd, int *exp This function tries to create a contiguous NumPy array of type typecode from an arbitrary python object `obj`. This should work for any sequence object. The argument `expectnd` is the expected number of dimensions, ignored if <= 0. The argument `expectdims` is an array of expected dimensions, ignored if <= 0. This routine raises a `ValueError` exception if the underlying `PyArray_ContiguousFromObject` routine fails, if the array has a bad shape, if the extent of a given dimension doesn't match the specified extent. If `obj` is a contiguous `PyArrayObject*` then a reference is returned; if `obj` is a python sequence, then a new `PyArrayObject*` is created and returned.

Beyond the Provided Typemaps

There are many C or C++ array/NumPy array situations not covered by a simple `%include "numpy.i"`. Nevertheless, `numpy.i` may still 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 three-dimensional array as a function argument, you could cut-and-paste the appropriate two-dimensional typemap into your interface file. The modification for the third 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.

Acknowledgements

Many people have worked to glue SWIG and NumPy (and its predecessors Numeric and numarray) together. 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. Their work has made this end result possible.

Generated on: 2007-03-19 04:09 UTC. Generated by Docutils from reStructuredText source.