Testing the numpy.i Typemaps

Author:	Bill Spotz
Institution:	Sandia National Laboratories
Date:	6 April, 2007

Contents

Introduction

Writing tests for the `numpy.i` SWIG interface file is a combinatorial headache. At present, 12 different data types are supported, each with 23 different argument signatures, for a total of 276 typemaps supported "out of the box". Each of these typemaps, in turn, might require several unit tests in order to verify expected behavior for both proper and improper inputs. Currently, this results in 1,020 individual unit tests that are performed when `make test` is run in the `numpy/docs/swig` subdirectory.

To facilitate this many similar unit tests, some high-level programming techniques are employed, including C and SWIG macros, as well as python inheritance. The purpose of this document is to describe the testing infrastructure employed to verify that the `numpy.i` typemaps are working as expected.

Testing Organization

There are three indepedent testing frameworks supported, for one-, two-, and three-dimensional arrays respectively. For one-dimensional arrays, there are two C++ files, a header and a source, named:

```
Vector.h
Vector.cxx
```

that contain prototypes and code for a variety of functions that have one-dimensional arrays as function arguments. The file:

```
Vector.i
```

is a SWIG interface file that defines a python module `Vector` that wraps the functions in `Vector.h` while utilizing the typemaps in `numpy.i` to correctly handle the C arrays.

The `Makefile` calls `swig` to generate `Vector.py` and `Vector_wrap.cxx`, and also executes the `setup.py` script that compiles `Vector_wrap.cxx` and links together the extension module `_Vector.so` or `_Vector.dylib`, depending on the platform. This extension module and the proxy file `Vector.py` are both placed in a subdirectory under the `build` directory.

The actual testing takes place with a python script named:

```
testVector.py
```

that uses the standard python library module `unittest`, which performs several tests of each function defined in `Vector.h` for each data type supported.

Two-dimensional arrays are tested in exactly the same manner. The above description applies, but with `Matrix` substituted for `Vector`. For three-dimensional tests, substitute `Tensor` for `Vector`. For the descriptions that follow, we will reference the `Vector` tests, but the same information applies to `Matrix` and `Tensor` tests.

The command `make test` will ensure that all of the test software is built and then run all three test scripts.

Testing Header Files

`Vector.h` is a C++ header file that defines a C macro called `TEST_FUNC_PROTOS` that takes two arguments: `TYPE`, which is a data type name such as `unsigned int`; and `SNAME`, which is a short name for the same data type with no spaces, e.g. `uint`. This macro defines several function prototypes that have the prefix `SNAME` and have at least one argument that is an array of type `TYPE`. Those functions that have return arguments return a `TYPE` value.

`TEST_FUNC_PROTOS` is then implemented for all of the data types supported by `numpy.i`:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- `float`
- `double`

Testing Source Files

`Vector.cxx` is a C++ source file that implements compilable code for each of the function prototypes specified in `Vector.h`. It defines a C macro `TEST_FUNCS` that has the same arguments and works in the same way as `TEST_FUNC_PROTOS` does in `Vector.h`. `TEST_FUNCS` is implemented for each of the 12 data types as above.

Testing SWIG Interface Files

`Vector.i` is a SWIG interface file that defines python module `Vector`. It follows the conventions for using `numpy.i` as described in the numpy.i documentation. It defines a SWIG macro `%apply_numpy_typemaps` that has a single argument `TYPE`. It uses the SWIG directive `%apply` as described in the numpy.i documentation to apply the provided typemaps to the argument signatures found in `Vector.h`. This macro is then implemented for all of the data types supported by `numpy.i`. It then does a `%include "Vector.h"` to wrap all of the function prototypes in `Vector.h` using the typemaps in `numpy.i`.

Testing Python Scripts

After `make` is used to build the testing extension modules, `testVector.py` can be run to execute the tests. As with other scripts that use `unittest` to facilitate unit testing, `testVector.py` defines a class that inherits from `unittest.TestCase`:

```
class VectorTestCase(unittest.TestCase):
```

However, this class is not run directly. Rather, it serves as a base class to several other python classes, each one specific to a particular data type. The `VectorTestCase` class stores two strings for typing information:

self.typeStr A string that matches one of the `SNAME` prefixes used in `Vector.h` and `Vector.cxx`. For example, `"double"`.

self.typeCode A short (typically single-character) string that represents a data type in numpy and corresponds to `self.typeStr`. For example, if `self.typeStr` is `"double"`, then `self.typeCode` should be `"d"`.

Each test defined by the `VectorTestCase` class extracts the python function it is trying to test by accessing the `Vector` module's dictionary:

```
length = Vector.__dict__[self.typeStr + "Length"]
```

In the case of double precision tests, this will return the python function `Vector.doubleLength`. We then define a new test case class for each supported data type with a short definition such as:

```
class doubleTestCase(VectorTestCase):
    def __init__(self, methodName="runTest"):
        VectorTestCase.__init__(self, methodName)
        self.typeStr  = "double"
        self.typeCode = "d"
```

Each of these 12 classes is collected into a `unittest.TestSuite`, which is then executed. Errors and failures are summed together and returned as the exit argument. Any non-zero result indicates that at least one test did not pass.