A Guided Example and Reference for wrapping C++ code for Python using SWIG

Sean Lewis

September 16, 2013

Contents

1 Introduction

1.1 Motivation

Python has gained significant popularity as a programming language in recent years. In particular, Python has been adopted by much of the scientific community thanks to its ease of use and its large package ecosystem. The biggest problem is when efficient computation is needed. One solution is write the code responsible for bottlenecks in C or C++ and generate wrapper code so it can be used in Python. SWIG is a tool to generate wrapper code that uses the Python C API so that calling a function in Python ends up calling the function in a compiled C/C++ library.

SWIG can wrap basic C code as well as mind-boggling C++ code. It can also generate wrappers for a wide variety of target languages. Thus, it can be difficult to find some information when dealing with a specific setup. This document should provide some help with the specific case of interfacing C++ code with Python.

This is a stand-alone document that provides an introduction to wrapping C++ code for Python. The full source code of the project may be found at `https://github.com/splewis/statistics-swig-example`.

1.2 Reference Material

- The Official SWIG reference guide is an extremely useful source of information: `http://www.swig.org/Doc2.0/Contents.html#Contents`
- SWIG is hosted on GitHub at `https://github.com/swig/swig`. The examples folder, `https://github.com/swig/swig/tree/master/Examples` is very useful.
- The SWIG mailing lists are also useful for tracking down errors: `http://www.swig.org/mail.html`.

1.3 Environment

Many of the commands/build systems depend upon common Unix-like system features. However, the final product (which consists of a python setup file, some C++ sources, and the SWIG-produced wrapper file) is perfectly usable on any system that has a C++ compiler and that Python's distutils tool supports. I frequently use C++11 features, so this example project requires a relatively recent C++ compiler (gcc 4.7 should do the trick).

1.4 C++ Code

- InvalidParameter - an exception class used for when passed arguments are out of range or invalid in any way.
- ProbabilityDistrubiton - abstract base class for a continuous probability distribution with 4 pure virtual functions.
- GaussianDistribution - implementation of a ProbabilityDistribution for a normal distribution.
- ExponentialDistribution - implementation of a ProbabilityDistribution for a exponential distribution.
- MathFunctions - common constants (π, $\sqrt{2}$) and useful functions (isprime).
- Regressions - functions for performing a regression from a vector of double pairs.
- DiscreteStats - functions for computing the mean/variance of a vector of doubles.

2 SWIG Wrapper Code

2.1 Include Statements

The core of the SWIG interface files is the include statements. Here is an example of an early interface file:

```
%module PyStatistics %{
    // Standard includes.
    // You should include all source files here.
    #include "../DiscreteStats.hpp"
    #include "../ExponentialDistribution.hpp"
    #include "../GaussianDistribution.hpp"
    #include "../InvalidParameter.hpp"
    #include "../MathFunctions.hpp"
    #include "../ProbabilityDistribution.hpp"
    #include "../Regressions.hpp"
%}

// Wrapper include statements:
// Only add things we actually want wrapped here.
%include "../InvalidParameter.hpp"
%include "../MathFunctions.hpp"
%include "../DiscreteStats.hpp"
%include "../Regressions.hpp"
%include "../ProbabilityDistribution.hpp"
%include "../ExponentialDistribution.hpp"
%include "../GaussianDistribution.hpp"
```

So what's wrong with this? Running swig on this file will yield several errors. It will also give a somewhat useless module. To get more useful we have to do some of the harder stuff below.

SWIG directives start with a %. The include and module directives are used above.

2.2 Exceptions

Cleanly handling exceptions is challenging. The factorial method, as implemented below, has this behavior:

```
long factorial(const int n) {
    if (n < 0)
        throw InvalidParameter("The factorial function cannot accept negative inputs.");
    long product = 1L;
    for (int i = 2; i <= n; i++)
        product *= i;
    return product;
}
```

```
>>> import PyStatistics as PS
>>> PS.factorial(-1)
terminate called after throwing an instance of 'statistics::InvalidParameter'
Aborted
```

Note that this terminate call will result in an **uncatchable exception**.

However, if we add an exception specifier, we get a much nicer behavior:

```
long factorial(const int n) throw(InvalidParameter) {
    if (n < 0)
        throw InvalidParameter("The factorial function cannot accept negative inputs.");
    long product = 1L;
    for (int i = 2; i <= n; i++)
        product *= i;
    return product;
}
```

```
>>> import PyStatistics as PS
>>> PS.factorial(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PyStatistics.InvalidParameter
```

As you can see, adding the exception specifier gave SWIG enough information to let us at least catch the exception. Unfortunately, exception specifiers are a horrible feature of C++ and I implore you to never use them.

Another way to handle exceptions is to wrap them to SWIG defined exceptions that end up as standard Python exceptions. It has the benefit of not relying on any Python-specific feature. The following code would be inserted in the SWIG interface file:

```
%exception {
    try {
        $action
    } catch (statistics::InvalidParameter &e) {
        std::string s("PyStatistics error: "), s2(e.what());
        s = s + s2;
        SWIG_exception(SWIG_RuntimeError, s.c_str());
    } catch (...) {
        SWIG_exception(SWIG_RuntimeError, "unknown exception");
    }
}
```

For this approach, available exceptions are:

```
SWIG_MemoryError
SWIG_IOError
SWIG_RuntimeError
SWIG_IndexError
SWIG_TypeError
SWIG_DivisionByZero
SWIG_OverflowError
SWIG_SyntaxError
SWIG_ValueError
SWIG_SystemError
```

Finally, I present my preferred approach. It is specific to Python, but ends up throwing the wrapped Python class of the original C++ exception. It should be inserted before the functions/methods that use the exception are included or declared in the SWIG interface file. It requires no exception specifiers.

This approach is adapted for readability from `http://stackoverflow.com/questions/15006048/`. It lends itself to macros very well.

```
%exception {
    try {
        $action
    } catch (statistics::InvalidParameter &e) {
        auto copy = static_cast<const statistics::InvalidParameter&>(e);
        auto instanceObj = new statistics::InvalidParameter(copy);
        auto ptrObj = SWIG_NewPointerObj(instanceObj,
                                         SWIGTYPE_p_statistics__InvalidParameter,
                                         SWIG_POINTER_OWN);
        SWIG_Python_Raise(ptrObj,
                          "InvalidParameter",
                          SWIGTYPE_p_statistics__InvalidParameter);
        SWIG_fail;
    }
}
```

2.3 Templates

Several functions used vectors as parameters or return types. Thankfully it's fairly easy to give a Pythonic wrapper so nobody has to write C++ in Python. Adding the below is enough to get a wrapper for a vector of doubles that can be manipulated in Python.

```
%include "std_vector.i"
%rename(append) std::vector<double>::push_back(double x);
%template(vector_double) std::vector<double>;
```

Now we have a vector of doubles that behaves almost the same as a Python list - except it only accepts doubles. We can create an object of this type from Python and pass it to any C++ function that accepts `std::vector<double>`.

Some functions in Regressions.hpp use a `std::vector<std::pair<double, double>>` as their argument. Wrapping this is no different - but to make it work cleanly we have to make some design decisions about the API. Since they were being used to represent 2-D points I chose to name them similarly. I also added a helper function named `addpoint` to the wrapper of `std::vector<std::pair<double, double>>`. This function lets me avoid ever having to create a pair object (though I can using the datapoint wrapper also created) and just add the X and Y values at once.

The %extend directive is very useful for adding wrapper functionality.

```
%include "std_vector.i"
%include "std_pair.i"

%rename(append) std::vector<std::pair <double, double> >::push_back(std::pair<double, double> x);

%template(datapoint) std::pair<double, double>;
%template(datalist) std::vector<std::pair <double, double> >;

%extend std::vector<std::pair <double, double> > {
    void addpoint(const double x, const double y) {
        auto pair = std::pair<double, double>(x, y);
        $self->push_back(pair);
    }
}
```

2.4 The Little Things

What makes a good API? Making a Python programmer use `std::vector<double>` is certainly not a good design decision. That's why we got rid of `push_back` and renamed it to `append`. When targeting python you should keep in mind the hidden methods and functions that make Python so clear and succinct. Among them are `__iter__`, `__str__`, and `__len__`.

The standard SWIG library includes the file `std_vector.i`, which did a lot of that work for our vector wrappers. The GaussianDistribution class can only get string output using a C++ ostream. However, using streams in Python the way they are used in C++ is a scary idea. It's much better to add a little wrapper code to let the str function on a GaussianDistribution object call that stream operator to get the string output. The %extend directive comes in handy again. Remember that using the print statement (or print function in Python 3 and onwards) calls `__str__` on the object.

```
%extend statistics::GaussianDistribution {
    std::string __str__() {
        std::ostringstream os;
        os << *($self);
        return os.str();
    }
}
```

```
>>> import PyStatistics as PS
>>> dist = PS.GaussianDistribution(0.0, 1.5)
>>> print dist
Gaussian distributed variable with:
  Mu: 0.000000
  Sigma: 1.500000
  E(X): 0.000000
  Var(X): 2.250000
```

3 Build Systems and Distribution

3.1 The Developer's View

The only special command a developer must run is SWIG itself.

```
swig -python -c++ src/bindings/PyStatistics.i
```

Once this is done, the developer can install the package the same way a user would install any typical python package that uses distutils with a setup.py file.

```
python setup.py install --prefix=~/.local
```

Since I assumed a Unix-like platform for the developer, there is a Makefile that runs these commands. Below is the setup.py file used:

```
#!/usr/bin/env python

from distutils.core import setup, Extension
from glob import glob


source_files = glob('src/*.cpp')
source_files.append('src/bindings/PyStatistics_wrap.cxx')
extension = Extension('_PyStatistics', source_files, language='c++')


setup(name='PyStatistics',
      author='Sean Lewis',
      author_email='splewis@utexas.edu',
      url='https://github.com/splewis/statistics-swig-example',
      description=('An EXAMPLE of wrapping some C++ code for Python using SWIG'
                   ' - this is not a statistics library!'),
      keywords='example SWIG C++ statistics',
      version='1.4.1',
      ext_modules=[extension],
      package_dir = {'': 'src/bindings'},
      license='MIT',
      py_modules=['PyStatistics'])
```

3.2 Source Distributions

Of course, there are problems with just having a bare source repository. Given that distutils and a setup.py file was used for our packaging, we can easily create a source distribution that will include the SWIG-generated file, eliminating that dependency for users while still being cross-platform.

You can generate a source archive by running:

```
python setup.py sdist
```

Uploading to the Python Package Index (PyPI) is also trivial. Just run:

```
python setup.py register
python setup.py upload
```

This uploads the same file as the earlier sdist command. (well, the contents should be the same - the file gets regenerated)

This package is uploaded to `https://pypi.python.org/pypi/PyStatistics`.

3.3 Binary Distributions

I only have personal experience working with and Debian-based platforms, so that is all I describe. Distutils has built-in commands for generating a rpm file or msi Windows installer.

Since the binary isn't really meant for use, but for demonstration, I only provide a 64-bit debian binary.

3.3.1 Debian

A useful package for this is `https://pypi.python.org/pypi/stdeb`. Install the stdeb package and run (you may need other debian packages - read the output of errors):

```
python setup.py sdist
cd dist
py2dsc PyStatistics-1.y.z.tar.gz [or whatever the current version is]
cd deb_dist/pystatistics-1.x.y
dpkg-buildpackage -rfakeroot -uc -us
```

Of course, you will need to do this on each platform you support.

3.4 Other Notes

Distutils isn't my favorite piece of Python, it can be problematic and troublesome. In particular, it's C++ compiler support is lacking. It enjoys passing strange arguments that don't make sense to C++ compilers that produces many warnings.

Distutils also has some support for SWIG. Don't use it. My experience has been that the distutils folks consider it untested and the SWIG folks have always expected projects to package the SWIG-generated files when releasing user source distributions.

Distutils also fails to parallelize the compile process, which can make it painfully slow. I've had good success with CMake to build SWIG modules. One approach is to use both, with CMake as the developer tool and distutils/binaries as the user tool. This way has the advantage of giving speed to the developer, and not requiring that users have SWIG installed.

4 Common Problems

4.1 ImportError: Undefined Symbol

The most common error I have faced in my experience occurs when importing the wrapped package:

```
ImportError: /home/splewis/.local/lib/python2.7/site-packages/_PyStatistics.so:
undefined symbol: _ZN10statistics7isprimeEi
```

To handle this, first demangle the symbol that is undefined using `c++filt`:

```
splewis@splewis-laptop $ c++filt _ZN10statistics7isprimeEi
statistics::isprime(int)
```

There are 2 common reasons this error may occur:

1. The isprime function is declared in a header, but never implemented
2. The implementation is missing for the C++ source files in the build system

5 Other Notes

5.1 Doxygen, Docstrings, and SWIG

Another aspect that makes Python so fun to write is the generous use of docstrings to easily find documentation. As of SWIG 2.0.7, SWIG has not yet implemented a way to read in Doxygen formatted comments in the C++ source and put them into Python docstrings, but it was a Google Summer of Code project for 2013, so it may be in future releases.

Here is the workaround I have used:

1. Use %feature("docstring", "1") at the top of your SWIG interface file.
2. Change your doxygen settings file to output xml files.
3. Use a script to parse the xml and paste it into SWIG interface files. A script named doxy2swig.py floats around the Internet but isn't hosted anywhere in particular. Searching by the filename should lead you to a working version of it.
4. Use the %include directive on each of the output files.

You may want to integrate this process into your build system. Additionally, I have written all the include statements (for the docstring files) into a auto-generated file that I include into the master SWIG interface file. This has the advantage of the source only assuming the existence of 1 auto-generated file.

5.2 Namespace Pollution

In a larger project, one may run into some namespace pollution issues. Be aware that you can separate files into different SWIG modules (which will yield separate Python modules). One can view the namespace by trying to use tab-completion in an IPython s

```
In [1]: import PyStatistics as PS

In [2]: PS.
PS.E
PS.ExponentialDistribution
PS.ExponentialDistribution_swigregister
PS.GaussianDistribution
PS.GaussianDistribution_swigregister
PS.InvalidParameter
PS.InvalidParameter_swigregister
PS.LINEAR
PS.PI
PS.ProbabilityDistribution
PS.ProbabilityDistribution_swigregister
PS.QUADRATIC
PS.SQRT2
PS.SwigPyIterator
PS.SwigPyIterator_swigregister
PS.approxfactorial
PS.cvar
PS.datalist
PS.datalist_swigregister
PS.datapoint
PS.datapoint_swigregister
PS.factorial
PS.isprime
PS.linreg
PS.mean
PS.polyeval
PS.primesuntil
PS.quadreg
PS.regression
PS.variance
PS.vector_double
```

```
PS.vector_double_swigregister
PS.vector_int
PS.vector_int_swigregister
```

Note that each class has a swigregister name also included. This may or may not be of concern, but there is a way to deal with it. Additionally, if you are wrapping a very large C++ library it might be wise to split the package into smaller modules or submodules. By creating several `__init__.py` files (each one would contain import statements from the SWIG-wrapped module) for each module you can manually divide and clean the namespace.

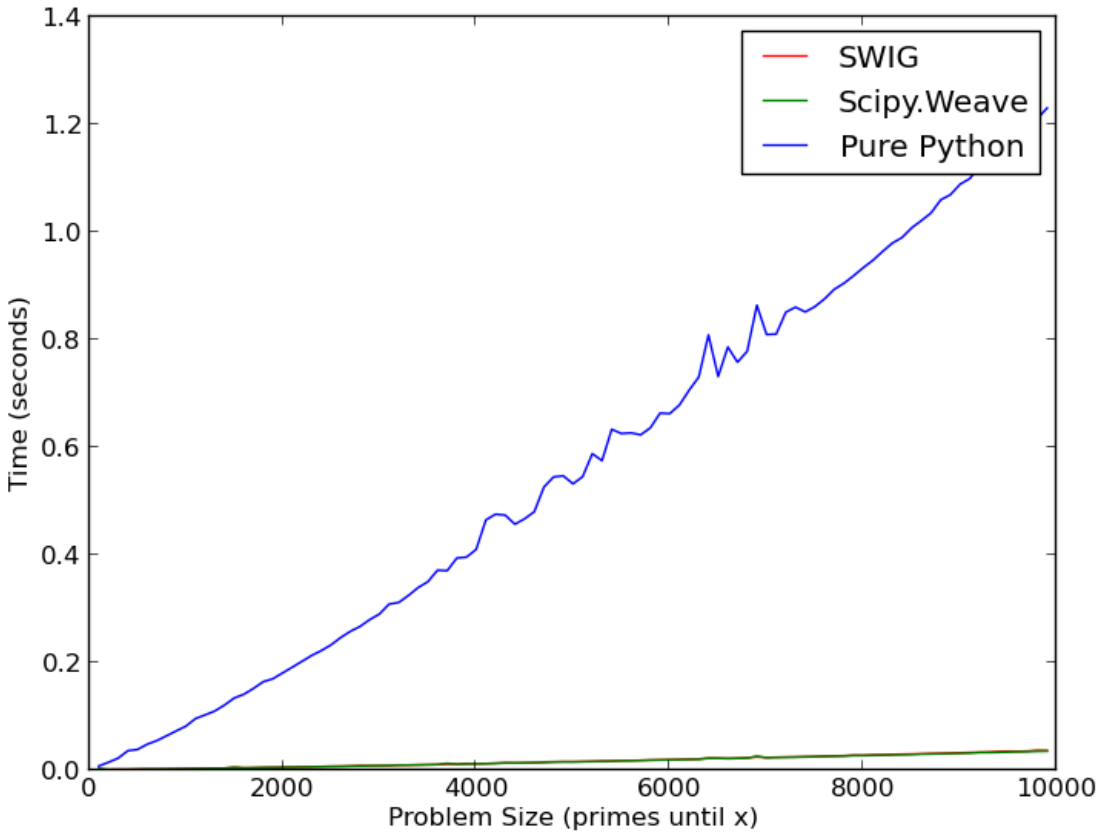
6 Alternatives and Benchmarks

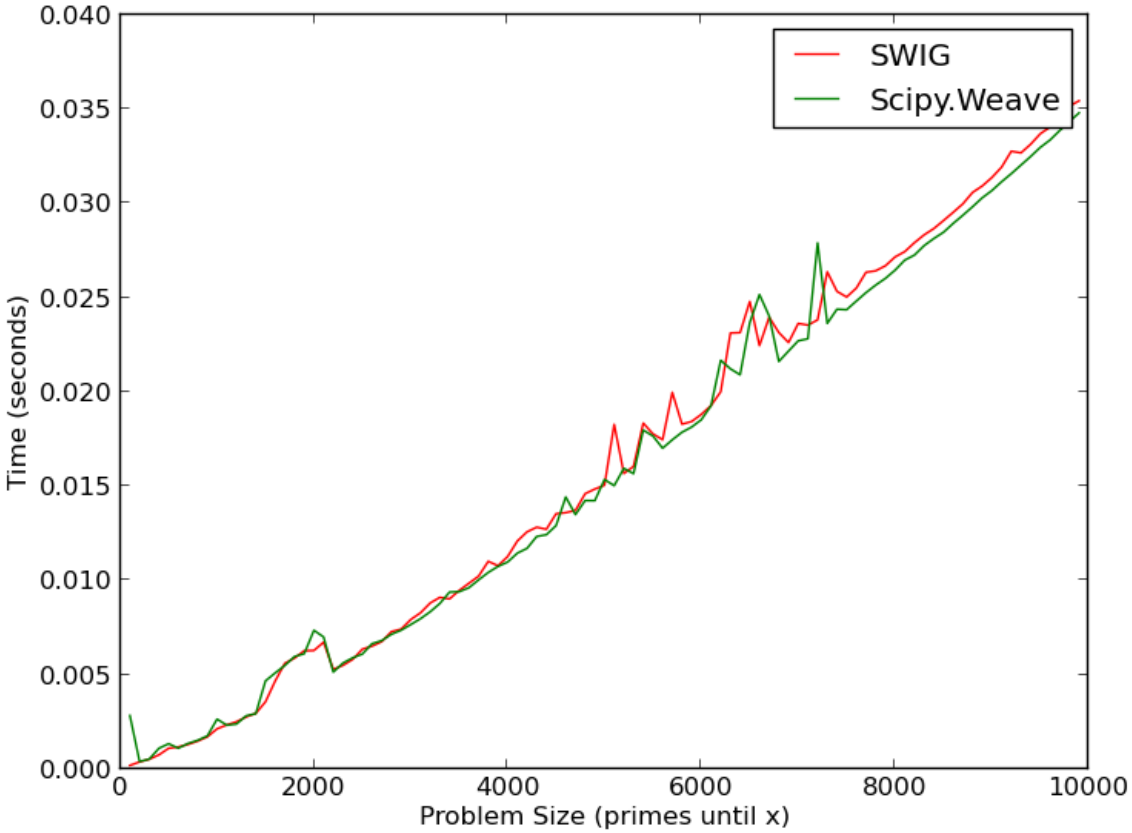
It's important to see the alternatives to using SWIG. I've had good luck with it, but other tools may fit your needs better. I've written a simple benchmark using the primesuntil function (which computes the primes up to a given number, not using any fancy sieves for simplicity).

This benchmark should only be used to get a rough idea about performance - it is very, very far from being thorough! The code for it is in benchmarks.py.

For wrapping a large C++ library, Boost.Python of the Boost project is worth a look. However, working with it can be tedious (in my opinion). Error messages can also be quite incomprehensible thanks to heavy template usage. Another option, if you only need to add speed to a Python library, is to inline some C/C++ code using scipy.weave, which compiles the code on the fly and eliminates the need to do any messy build system.

6.1 Results





As expected, the C++ accelerated code vastly outperforms the pure Python counterpart. It is interesting to note that the overhead given by SWIG seems to be fairly inconsequential.