On Embedding Gretl in a Python Module

Christine Choirat and Raffaello Seri

[1] Department of Quantitative Methods,
School of Economics and Business Management,
Universidad de Navarra, Pamplona (Spain)
cchoirat@unav.es
[2] Dipartimento di Economia,
Università degli Studi dell'Insubria , Varese (Italy)
raffaello.seri@uninsubria.it

Abstract. Additional functionalities can be developed for Gretl either directly in the main C code or with the Gretl scripting language. We illustrate through an example how it would be possible to wrap the C source of Gretl with SWIG to create an interface to Python that makes use of the matrix library NumPy. Such an interface would make it easier for users to extend Gretl since it would allow for developing and distributing Gretl extensions as Python modules.

Key words: Python, C, SWIG, Libgretl API

1 Introduction and motivation

To extend the functionalities of Gretl,[3] it is either possible to add them in the C source or to use the Gretl scripting language. The former option is only possible for users who have a very sound knowledge of C and who understand how the source of Gretl is structured. Besides, it makes it hard to share these added functionalities with the rest of the user base unless they are accepted for a next release. The latter option requires learning yet another field-specific language. Even if the speed benchmarks are good and the syntax easy, a program-specific macro language can never be as powerful as a full-featured scripting language (either domain-specific such as R, see [1, 2], or general such as Python, see [3]).

The scripting language that we have chosen to embed Gretl in is Python, which is free, open-source and available on many platforms. Python is indeed a very powerful, easy-to-learn and well-documented language with a very clear syntax. Writing and distributing documented Python modules is simple as well. Matrices are not a native Python data type. However, the very mature project NumPy[4] provides an efficient implementation of $N-$dimensional arrays (and

[3] See http://gretl.sourceforge.net/.
[4] See http://www.scipy.org/.

therefore matrices) accessible via its C API. Moreover, many other Python modules are of interest, such as the NumPy-based scientific library SciPy[5] or the plotting module matplotlib.[6] So, the objective is to get the Gretl embedding make use of NumPy.

In Section 2, we review the tools that can be used to embed C code in a Python module. We show that SWIG is the most powerful one (though not the easiest). Then in Section 3, we see how Gretl could be embedded. Section 4 concludes.

2 Possible implementation choices

Let us consider the following code (which is a simplified version of the f2py example found on the SciPy wiki).[7]

```
/* example1.c */
void func(int n, double *x) {
  int i;
  for (i=0; i<n; i++) {
    x[i] = x[i] + i;
  }
}
```

Python cannot be extended as trivially as R (in particular as can be done through the `.C` function, see Section 5.2 of the R manual [4]) and some nontrivial knowledge of the NumPy C API[8] is required even in the case of such a simple function (see *e.g.* Chapter 14 of [5]). Fortunately, two popular tools, namely f2py and SWIG, make extension development much easier.

2.1 f2py

f2py[9] is a utility that is primarily meant to wrap Fortran code as Python/NumPy modules. However, it seems to be the easiest way to wrap C code as well. It generates a documented Python module that works very smoothly with NumPy. The only required step is to write a signature file, which states that the function to be wrapped is a C function and provides the necessary information about its arguments:

[5] SciPy provides among others routines for statistics, optimization, numerical integration, linear algebra, Fourier transforms, signal processing, image processing, genetic algorithms, ODE solvers and special functions.

[6] See `http://matplotlib.sourceforge.net/`.

[7] See `http://www.scipy.org/Cookbook/f2py_and_NumPy`.

[8] See `http://docs.scipy.org/doc/numpy/reference/c-api.html`.

[9] See `http://www.scipy.org/F2py`.

```
! m1.pyf
python module m1
interface
  subroutine func(n,x)
    intent(c) func
    intent(c)
    integer intent(hide), depend(x) :: n=len(x)
    double precision intent(inplace) :: x(n)
  end subroutine func
end interface
end python module m1
```

Compilation is straightforward: `f2py m1.pyf example1.c -c` generates a Python extension module `m1.so` (on Linux, since the extension is platform-dependent). On the Python side, we get (the »> symbol indicates the Python prompt):

```
»> import numpy, m1
»> a = numpy.array([1., 3., 5.])
»> a
array([1., 3., 5.])
»> m1.func(a)
»> a
array([ 1.,  4.,  7.])
»> print m1.func.__doc__
func - Function signature:
  func(x)
Required arguments:
  x :  rank-1 array('d') with bounds (n)
```

However, as far as we know, no large-scale project as ever been wrapped with f2py, since a lot of manual work is required: an interface has to be written for every C function.

2.2 SWIG

On the other hand, SWIG[10] allows for an almost automatic wrapper generation (including such things as documentation generation and exception handling), the price to pay however is that integration with NumPy is not as easy as with f2py by default. SWIG stands for “Simplified Wrapper and Interface Generator”

[10] See `http://www.swig.org/`.

and allows for interfacing C (and C++) code with several target languages (C#, Java, Ocaml, Octave, Python, R, to name a few). It is successfully used in very large scale projects, one of the most famous being wxPython,[11] which provides Python bindings to the C++ cross-platform GUI library wxWidgets.[12]

For standard data types (`int`, `double`, ...), SWIG works very smoothly. Let us consider the following case, where we have a C file and its associated header file.

```
/* example2.c */
#include "example2.h"
int add1(int n) {
    return n + 1;
}
double add2(double x, double y) {
    return x + y;
}
```

and

```
/* example2.h */
int add1(int n);
double add2(double x, double y);
```

The only required step is to write an interface file `example2.i` (made of two main parts, the headers to be directly included and the functions to be parsed).

```
// example2.i
%module m2
%{
#define SWIG_FILE_WITH_INIT
// Includes the header in the wrapper code
#include "example2.h"
%}
// Parses the header file to generate wrappers
%include "example2.h"
```

and use SWIG to generate wrappers for the target language Python:

```
swig -python example2.i
```

[11] See `http://www.wxpython.org/`.
[12] See `http://wxwidgets.org/`.

A simple Python script

```
# setup_example2.py
from distutils.core import setup, Extension
setup(name="m2",
      ext_modules=[Extension("_m2",
                             ["example2.i", "example2.c"])])
```

takes care of building the extension module `m2`:

```
python setup_example2.py build_ext --inplace
```

From the Python interpreter, we get:

```
»> import m2
»> m2.add1(3)
4
»> m2.add2(3, 4.0)
7.0
```

2.3 SWIG and NumPy

Applying directly the method of Section 2.2 to `example1.c` (and the associated header `example1.h`) leads to a Python function that does not behave the expected way. If we build a module `m3` as above, we get:

```
»> import numpy, m3
»> a = numpy.array([1., 3., 5.])
»> m3.func(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func() takes exactly 2 arguments (1 given)
»> m3.func(3, a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: in method 'func', argument 2 of type 'double *'
```

The problem comes from the fact that SWIG does not know how to transform pointers into NumPy arrays. However, it is possible to define *typemaps*, which are SWIG procedures written in a C-like syntax, in order to make the transformation easier. Many of these typemaps are already available in the file `numpy.i` (which is part of the NumPy distribution). We can use it in the interface file `example1.i` to specify (with `%apply`) that the arguments of the function call are the length of the array and the array itself:

```
// example1.i
%module m4
%{
#define SWIG_FILE_WITH_INIT
#include "example1.h"
%}
%include "numpy.i"
%init %{
import_array();
%}
%apply (int DIM1, double* INPLACE_ARRAY1) {(int n, double *x)};
%feature("autodoc", "func(array x) -> x") func;
void func(int n, double *x);
```

We now get the expected behavior, together with a customized documentation (that could also be generated automatically):

```
»> import numpy, m4
»> a = numpy.array([1., 3., 5.])
»> m4.func(a)
»> a
array([ 1.,  4.,  7.])
»> print m4.func.__doc__
func(array x) -> x
```

3 SWIG and Gretl

As we have seen in Section 2, SWIG can be used to wrap a program as large as Gretl in an almost automatic way. Let us consider the minimalistic (and obviously very incomplete) case of three interface files (assuming the source code of Gretl is in `src/`), namely `libgretl.i`, `version.i` and `gretl.i`

```
// libgretl.i
%module gretl
%{
#include "src/libgretl.h"
%}
%include "src/libgretl.h"

// version.i
%module gretl
```

```
%{
#include "src/version.h"
%}
%include "src/version.h"

// gretl.i
%module gretl
%include "version.i"
%include "libgretl.i"
%{
#define SWIG_FILE_WITH_INIT
%}
```

The following setup script (hard-coding the directory of the XML library) generates a Python module `gretl`:

```
# setup_gretl.py
from distutils.core import setup, Extension
setup(name="gretl",
      ext_modules=[Extension("_gretl", ["gretl.i"],
      include_dirs=["/usr/include/libxml2/"])])
```

The module can then be called from Python:

```
»> import gretl
»> gretl.GRETL_VERSION
'1.8.0'
```

C structures are automatically transformed into Python classes. For example, the following structure (in `src/libgretl.h`)

```
typedef struct _cmplx cmplx;
struct _cmplx {
    double r;
    double i;
};
```

can be accessed from Python as

```
»> gretl._cmplx
<class 'gretl._cmplx'>
»> z = gretl._cmplx()
»> z.r
0.0
»> z.i
0.0
```

4 Conclusion and further developments

We have provided an illustration of the fact that it is possible to embed Gretl's functionalities into a high-level scripting language such as Python. At this point, it would be very fruitful to get some comments from the Gretl community.

- Which target language should be focused upon? Python is the easiest in terms of SWIG integration, but R has more built-in econometric functions and is of more widespread use. (Remark however that the Python package RPy[13] allows for using R from within Python. Moreover, the R community tends now to favor the use of the Rcpp[14] package to create bindings to large C/C++ libraries such as RQuantLib[15]).
- How far should we go in the use of SWIG? The source code of Gretl is large and complex. Gretl defines its own matrix library and matrix operations (in `src/gretl_matrix.c`), its own optimization functions (in `src/gretl_bfgs.c`), etc. All these tools (and many more) are available through NumPy (not to mention Python's XML standard library and the graphical library matplotlib). So, some help from the Gretl developers would allow for selecting the crucial parts to be wrapped.

Writing the proper SWIG typemaps (along the lines of the C++ machine learning library Shogun,[16] see [6]), it would be possible to build a Gretl module for R, Python, Octave and Matlab (and any other language supported by SWIG) with only a minimal effort.

[13] See `http://rpy.sourceforge.net/`.
[14] See `http://cran.r-project.org/web/packages/Rcpp/index.html`.
[15] See `http://cran.r-project.org/web/packages/RQuantLib/index.html`.
[16] See `http://www.shogun-toolbox.org/`.

Bibliography

[1] Cribari-Neto, F., Zarkos, S.: R: yet another econometric programming environment. Journal of Applied Econometrics **14**(3) (June 1999) 319–329

[2] Racine, J., Hyndman, R.: Using R to teach econometrics. Journal of Applied Econometrics **17**(2) (April 2002) 175 – 189

[3] Choirat, C., Seri, R.: Econometrics with Python. Journal of Applied Econometrics (forthcoming)

[4] R Development Core Team: Writing R extensions. `http://cran.r-project.org/doc/manuals/R-exts.html`

[5] Oliphant, T.: Guide to NumPy. `http://numpy.scipy.org/numpybook.pdf`

[6] Sonnenburg, S., Raetsch, G., Schaefer, C., Schoelkopf, B.: Large scale multiple kernel learning. Journal of Machine Learning Research **7** (July 2006) 1531–1565