Driving and Extending Legacy Codes using Python

Neilen Marais and David B. Davidson

Department of Electrical and Electronic Engineering
University of Stellenbosch, South Africa
nmarais@gmail.com, davidson@sun.ac.za

Abstract: Software Engineering aspects in Computational Electromagnetics (CEM) are becoming more important as the complexity of CEM codes continue to increase. Object orientated programming (OOP) methods promise to alleviate the challenges posed by more complex software systems, but offers little help for legacy codes. Python, an object-oriented very high level language (VHLL), can be used to extend legacy codes. It provides the dual benefit of a very productive programming environment and of enabling legacy codes to be migrated to object orientated designs with low risk. The application of this method is described in the context of eMAGUS, a microwave Finite Element Method code.

Keywords: Object Orientated Design, Python, Fortran, Finite Element Method

1 Introduction

The purpose of Computational Electromagnetics (CEM) is the numerical solution of EM problems, but the end product of CEM development itself is software. As such, good software engineering principles should be an important tool in the CEM worker's toolbox. However, the mathematical and electromagnetic insight required for CEM has traditionally overshadowed the importance of the software aspects.

On the other hand, the software complexity of CEM codes continues to increase thanks to several factors. The solution of progressively more general problems demand an increasing array of basis function types to be integrated. The full-wave solution of ever larger problems require more sophisticated field modelling and more effective solution strategies. The unsustainability of historic single CPU performance growth is forcing the adoption of parallel methods, along with the increased complexity they bring. At the same time, users are demanding better solution accuracy while geometries are simultaneously becoming more complex, necessitating more flexible geometry handling. These trends have lead to software engineering aspects becoming quite important to CEM workers.

In this paper, the use of Python to extend the microwave Finite Elements Method (FEM) code, eMAGUS, is discussed. eMAGUS and its predecessors has been developed by the Computational Electromagnetics Group (CEMAGG) [1] at the University of Stellenbosch over almost a decade, and is coded in Fortran 90. It is shown how existing code and data structures can be utilised and extended with Python, and what the advantages are.

2 CEM Development Practice

Most numerical computation has traditionally been done in Fortran, particularly Fortran 77 (F77), which is arguably archaic as far as language features are concerned. More recently, Fortran 90 (F90), C and C++ have become more popular. While F90 and C both have their respective advantages, neither are well suited to the use of more modern software paradigms. C++ is certainly capable in this regard, but presents a rather steep learning curve.

Regardless of a particular language's merits, it is of no use if existing code is written in another language. Investments in old code are often considerable. Even if a new language were guaranteed to boost productivity significantly, the cost, and perhaps more significantly, risk involved in either porting or re-implementing a code can be very unattractive.

More recently, the use of very high-level languages (VHLL) such as Python [2], Ruby [3] and Perl [4] have shown promise as a means of moving existing codes forward without taking the risk of a full re-implementation. At first glance, these languages may seem unattractive for numerical computation, since VHLL execution speed may be too slow. However, most VHLLs can interface with compiled libraries for speed critical loops, while some provide additional means of speeding up critical code sections.

3 Python and Numeric Computing

Python is a modern object orientated VHLL featuring exceptionally clear syntax and powerful language features. It is both easy to learn, and a very productive environment for expert users. Features like automatic memory management, high level data types and extensive, freely available software libraries allow very rapid development, and much shorter code than that of an equivalent program written in e.g. C++ or Fortran. This combined with its free and open source licencing makes Python attractive for most any task.

Python has always had a C language API, allowing the extension of Python modules such that it is transparent (i.e. appears to be part of the Python environment) to the Python user, at the expense of extra work for the C programmer. More recently, automatic Python wrapper generators such as SWIG [5] and F2PY [6] have made it quite painless to extend Python using new or existing code written in compiled languages such as C, C++ and Fortran.

Python's ability to wrap existing code led to standard computational libraries such as LAPACK [7] and FFTPACK [8] along with fast array and matrix handling becoming available through the Numerical Python [9] package for Python. The Numerical Python work has since been merged and extended by the SciPy [10] project. An active developer and user community has built around SciPy.

Building on SciPy, the iPython [11] interactive shell and the Matplotlib [12] Pylab mode make for comfortable interactive use in the style of MATLAB (R), while keeping the full capabilities of a general language and its libraries available. The recent addition of iterative and direct sparse matrix solvers, as well as a PETSc [13] library interface to SciPy makes it well suited for FEM codes.

```
MODULE DATA
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE DATA

MODULE prog
  USE data
  IMPLICIT NONE
CONTAINS

  SUBROUTINE init_data()
    ! Initialise the array
    test_arr = (/1,2,3,4,5/)
  END SUBROUTINE init_data

  SUBROUTINE process_data(factors, n)
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i
    DO i = 1,n
       test_arr = test_arr - (-1)**i &
                  * test_arr**factors(i)
    END DO
  END SUBROUTINE process_data

END MODULE prog
```

Figure 1: Example F90 code

4 Wrapping Fortran 90

F2PY is an automated python wrapper generator for F77 and F90. It has comprehensive support for F77 language features. It can automate many of the low level details F77 subroutine calls usually require, such as determining the size of arrays. It also supports F90 module data and subroutines, but does not have direct support for derived types and assumed size dummy arguments at this time. Both limitations can be overcome by the addition of some simple wrappers to the Fortran code.

Assuming iPython and F2PY are set up, and that Intel's Fortran compiler is being used in a UNIX environment, the code in Fig. 1 can be compiled into a Python module by issuing:

```
$ f2py --fcompiler=intel -m testmod -c test_data.f90 test_prog.f90
```

The `-m test` flag specifies the python module name as `testmod`. The `-c test_data.f90 test_prog.f90` flag specifies the Fortran source files to be wrapped and compiled. Now we can manipulate the data and call the Fortran routines from the iPython shell.

```
$ ipython -pylab
...
  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
In [3]:testmod.prog.init_data()
In [4]:testmod.data.test_arr
Out[4]:[ 1., 2., 3., 4., 5.,]
In [5]:testmod.prog.process_data([1,2])
In [6]:testmod.data.test_arr
Out[6]:[ -2.,-12.,-30.,-56.,-90.,]
```

`In [x]:` represents the x'th user input, and `Out [x]:` the x'th output from the iPython shell. `In [1]:` loads the python module created by F2PY. `In [2]:` causes the value to be printed. Note how a new namespace is assigned for each module, in contrast to the F90 norm of importing all symbols into the current namespace. `In [5]:` shows how the length parameter is automatically passed to the Fortran routine.

5 Extending eMAGUS

eMAGUS (formerly known as FEMFEKO) is a research microwave FEM code developed by the CEMAGG group at Stellenbosch University over the last decade. It is chiefly used for research in FEM methods, and is coded in F90. eMAGUS contains many well tested routines that can be re-used when trying new techniques, though its code structure is somewhat specialised to frequency domain work.

Wrapping eMAGUS with F2PY provides several immediate benefits:

- Python's extensive use of namespaces allows the re-use of eMAGUS routines without dictating the structure of the new code.
- New methods are developed much faster owing to the use of a VHLL.
- Interactive simulation and plotting with scripting capabilities become a possibility.
- Readily available Python libraries can be used for many purposes, like plotting, model visualisation and data IO in various formats.
- The availability of several matrix solution modules with minimal extra coding requirements.

The present author is undertaking research into Finite Element Time Domain (FETD) methods [14]. F2PY wrappers are used to access the mesh processing and curl-conforming basis function and field-reconstruction routines from eMAGUS. New code is written using the core SciPy array data-types and numerical libraries.

6 Moving eMAGUS Forward

Applying object orientated (OO) design methods to eMAGUS is becoming increasingly attractive as its complexity increases, but F90 lacks the language features to do so cleanly. One option would be a rewrite from scratch, but such a move is always risky. Language wrappers provide a way of moving forward without the associated risk. The Python language features can be used to encapsulate the existing program structure in an OO framework piece by piece. New features are written in Python using the OO structure. As it becomes necessary to extend their functionality, original F90 routines can be replaced with Python equivalents. Re-implementing routines piecemeal considerably eases their testing, since there are known-good implementations to compare with.

Performance critical code sections can be kept in F90, or written in another compiled language at only a small cost in flexibility. Since the overall program structure would be defined in Python, the effect of specialised compiled modules on the overall design will be localised. Another possibility is to treat the Python code of performance critical sections using tools like Weave [15] and Pyrex [16].

Following this method, the code is gradually moved to a flexible and easily extensible OO design. Useful code built up over the years can be re-used, and at no time will the code be unusable. This means new development and research could happen in parallel to the modernising of the code structure, avoiding the case of two divergent code trees. Using a distributed version control system such as GNU Arch [17] to keep track of simultaneous development may also be helpful.

7 Conclusion

This paper has discussed Python as a tool for CEM code development. The use of Python for the rapid development of CEM codes, and the use of F2PY to incorporate legacy Fortran code in an object orientated design was detailed, and the advantages discussed. These abilities are very useful in the context of conducting research in the rapidly evolving field of CEM research.

References

[1] "University of Stellenbosch Computational Electromagnetics Group home page," Available: http://research.ee.sun.ac.za/cem/index.html.

[2] G. van Rossum *et al.*, "Python," Available: http://www.python.org.

[3] Y. Matsumoto *et al.*, "Ruby," Available: http://www.ruby-lang.org/en/.

[4] L. Wall *et al.*, "Perl," Available: http://www.perl.org/.

[5] D. Beazley *et al.*, "Simplified Wrapper and Interface Generator (SWIG)," Available: http://www.swig.org/index.html.

[6] P. Peterson, "F2PY: Fortran to Python interface generator," Available: http://cens.ioc.ee/projects/f2py2e/.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, or http://www.netlib.org/lapack/lug/lapack_lug.html: Society for Industrial and Applied Mathematics, 1999.

[8] P. N. Swarztrauber, "FFTPACK," Available: http://www.netlib.org/fftpack/.

[9] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, *et al.*, *Numerical Python*, http://numeric.scipy.org/numpydoc/numdoc.htm.

[10] "SciPy - Scientific tools for Python," Available: http://www.scipy.org/.

[11] F. Pérez, "iPython: An enhanced Interactive Python shell," Available: http://ipython.scipy.org/.

[12] J. Hunter *et al.*, "Matplotlib," Available: http://matplotlib.sourceforge.net/.

[13] "Portable, Extensible Toolkit for Scientific Computation (PETSc)," Available http://www-unix.mcs.anl.gov/petsc/petsc-as/.

[14] J.-F. Lee, R. Lee, and A. Cangellaris, "Time-domain finite-element methods," *IEEE Trans. Antennas Propagat.*, vol. 45, no. 3, pp. 430–442, Mar. 1997.

[15] E. Jones, "Weave," Available: http://www.scipy.org/documentation/weave/.

[16] G. Ewing, “Pyrex - a Language for Writing Python Extension Modules,” Available: http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.

[17] T. Lord, “GNU Arch,” Available: http://www.gnuarch.org/.