F2PY: A tool for connecting Fortran and Python programs

Article *in* International Journal of Computational Science and Engineering · January 2009

DOI: 10.1504/IJCSE.2009.029165

CITATIONS
180

READS
3,442

1 author:

Pearu Peterson
Quansight
61 PUBLICATIONS **8,194** CITATIONS

SEE PROFILE

F2PY: a tool for connecting Fortran and Python programs

Pearu Peterson

Centre for Nonlinear Studies,
Institute of Cybernetics at Tallinn Technical University,
Akadeemia tee 21, 12918 Tallinn, Estonia
E-mail: pearu.peterson@gmail.com

Abstract: In this paper we tackle the problem of connecting low-level Fortran programs to high-level Python programs. The difficulties of mixed language programming between Fortran and C are resolved in an almost compiler and platform independent way. We provide a polished software tool F2PY that can (semi-)automatically build interfaces between the Python and Fortran languages and hence almost completely hide the difficulties from the target user: a research scientist who develops a computer model using a high-performance scripting approach.

Keywords: high-performance scripting; Fortran; Python; code generation; F2PY.

Reference to this paper should be made as follows: Peterson, P. (2009) 'F2PY: a tool for connecting Fortran and Python programs', *Int. J. Computational Science and Engineering*, Vol. 4, No. 4, pp.296–305.

Biographical notes: Pearu Peterson is a Senior Researcher in the Laboratory of Systems Biology at the Institute of Cybernetics at the Tallinn University of Technology, Tallinn, Estonia. In 2001 he received his PhD in Natural Sciences from the Tallinn University of Technology for the study on multi-soliton interactions. His research interests include solitons, nonlinear integrable PDEs, water waves, hydrodynamics, scientific computing, heart modelling, and confocal and florescence microscopy. He is co-author to many open-source projects such as SciPy that focus on scientific computing within Python.

1 Introduction

Developing computer models is a natural part of scientific research. These models typically combine components of mature scientific software of varying levels of complexity. Using low-level software libraries leads to a better performance when running the computer model, but slows down and distracts from model development because of the need to deal with software related technical details. High-level interactive programming environments such as Python simplify and speed up development of computer models but the speed of solving the model may be unacceptable for many research problems. Since the developers of computer models are often scientists with no training in the development of software (Wilson, 2006), they prefer to focus on solving scientific problems rather than software problems. This leads to the dilemma of how to best enable non-professional programmers to develop high-performance and robust computer models efficiently. One solution to this difficult software problem is to use interfaces to low-level high-performance software in a high-level programming environment (Ousterhout, 1998).

This paper describes a solution based on combining low-level Fortran code with the high-level Python programming environment. F2PY (a Fortran to PYthon interface generator) was specifically designed to create Python wrappers to large Fortran libraries (Peterson et al., 2001), and has already gained substantial popularity: currently F2PY provides complete support for wrapping Fortran 77 codes and partial support for wrapping Fortran 90 or newer codes. By interfaces we mean abstract descriptions of how Fortran subprograms should be accessible from Python, or vice-versa. Wrappers are implementations of such interfaces.

This paper describes the developments of F2PY, and provides examples of possible ways that F2PY could be used for scientific research. Note that we refer to Fortran 77, Fortran 90, Fortran 95, and Fortran 2003 standards simply as Fortran and use the full name only when particular details demand. Although F2PY was initially developed to wrap Fortran 77 programs to Python, it currently supports also newer Fortran standards with the extend that will be discussed in this paper.

Fortran has been the standard language of choice for scientific computing for more than three decades. Python is becoming an increasingly popular choice for scientific computation because of its many features that are attractive for scientists. Python is a scripting language that has a very clean and easy-to-learn syntax;

Python supports a very high-level object-oriented programming paradigm with many predefined data types and useful modules in the standard library; Python is easy to extend and its programs are highly portable. High-quality scientific computational packages in Python have emerged within the last ten years (Oliphant, 2007). For example, NumPy offers most of the basic functionality of Matlab, and SciPy provides a rich user-friendly scientific computing environment with interfaces to different high-performance software libraries written in C/C++ and Fortran (LAPACK, for instance).

The task of creating interfaces between programming environments such as Fortran and Python is nontrivial due to both the complexity of the low-level application programming interfaces, that often contain many technical details, and the extra knowledge regarding mixed-programming techniques between both Fortran and C, and C and Python. Finding a portable and compiler independent way of accessing some Fortran constructions from C can require a considerable amount of research and resources. For a scientific programmer, the threshold of learning all of the required technical details and resolving different portability issues, is prohibitively high. As a result, the developer often chooses to program everything either in a low-level language or in a high-level language, with a resultant computational model that is often far from optimal.

The aim of developing F2PY was to find an optimal solution to the Fortran and Python connection problem. The design criteria were that the programming solution should be appropriate and easy to use both for a scientist, who occasionally needs to use a Fortran subprogram from Python, and for a Python package developer, who needs to provide flexible and Python friendly interfaces to large Fortran libraries. The results of this work will be important for scientific programming groups that have developed large Fortran libraries over decades and now need to increase both the development productivity of their products and widen their code base to new users.

The following general design criteria describe the optimal software solution:

- No prior knowledge of mixed-programming techniques should be required to create and to use the interfaces between Python and Fortran programs.
- The development of a Fortran library of subprograms and a Python program using that library, should be independent. That is, neither the Fortran nor the Python user need not to be familiar with the other language.
- Creating robust and immediately usable wrappers between Fortran and Python programs should be automatic and triggered by a single command. However, there should also be an opportunity to tune the interfaces.
- The interface generator tool should take advantage of the information available in the Fortran source code and create as *Pythonic*[1] interface as possible.
- The interfacing solution should be as easy to use for large Fortran libraries (with thousands of subprograms) as for a simple Fortran procedure (Fortran function or subroutine).

A number of tools exist that simplify connecting C/C++ to Fortran for wrapping Fortran (77/90/95/2003) programs to Python like SWIG (Beazley, 2003), SIP, Boost.Python, Babel, and others. However, not one of them satisfies the single optimal solution criterion stated above, even in the case of simple Fortran procedures. To use these tools one needs, as a minimum, to create a tool-dependent specification file containing C/C++ declarations from which wrapper codes can be generated. When interfacing Fortran code, one needs to write pseudo-declarations for Fortran procedures that describe them as if they are C functions. Such an approach requires an extensive knowledge of both Fortran-C mixed-programming techniques and compiler specific issues, especially when interfacing Fortran 90 or newer subprograms and data collections. For example, some Fortran 90 compilers have rather poor support for mixed-programming with C, thus it is impossible to access Fortran module subprograms from C, see Section 2.4. Very few scientists have the experience required to create such interfaces in an efficient way using these tools.

The following example illustrates the F2PY solution to the Fortran and Python connection problem: with a single command we can create Python wrappers to all the 1301 LAPACK functions (the `sh>` line indicates a command in a Unix or DOS terminal window, \ is line continuation):

```
sh> f2py -m lapack LAPACK-3.0/SRC/*.f -c \
     --link-blas_opt
```

and have them immediately usable from Python:

```
sh> python
>>> from numpy import array, float64
>>> from lapack import dgesv
>>> # Solve a system of linear equations A*x=b
>>> A = array([[1,2], [3,4]])
>>> b = array([[5], [6]], float64,\
              order='FORTRAN')
>>> # see dgesv.__doc__ for argument information
>>> dgesv(2,1,A,[0,0],b,0)
>>> print b # the solution x is stored in b
[[-4. ]
 [ 4.5]]
```

To illustrate how much work F2PY can save a developer, note that the source file for LAPACK Fortran-Python bindings has about 290,000 lines of C code and it takes about 10 min on a standard 2007 laptop to generate and to compile all sources. If the same task would be carried out by an average C programmer, it would require about

six years to complete (assuming that the productivity is about 25 lines of code per hour (Prechelt, 2000)).

This paper is organised as follows. First, we present an overview of the task of wrapping Fortran procedures to Python. This will also include an original idea of overcoming the difficult problem of accessing Fortran 90 symbols from C in a compiler independent way. Next we show the use of F2PY as an automatic Fortran-Python wrapper generation tool. Following this is a short tutorial and feature overview, including F2PY usage strategies, interface peculiarities, and limitations. Finally conclusions and future work plans will be discussed.

2 Wrapping Fortran programs to python

A Python wrapper of a Fortran procedure (a Fortran `subroutine` or `function`) is an adaptor-like function (implemented as Python extension function) that converts Python objects (function arguments) to objects that can be used as Fortran procedure arguments, makes the call to the Fortran procedure, and finally, converts the results (changed arguments and return values) to Python objects.

The connection between Fortran and Python is achieved via C. While Python and C interoperability is well defined by the Python/C API then there are two major issues when creating Python wrappers to Fortran procedures:

- Fortran and C interoperability was not standardised at all until Fortran 2003 was published. This causes a number of portability issues that are related to compiler dependent name mangling and different binary representations of data types (e.g., different array storage order in C and Fortran, different array dope vectors when using different Fortran compilers, etc.). Resolving these portability issues sometimes require the usage of complex wrapping techniques.
- There are differences in Fortran and Python/C languages such as the lack of direct counterparts of certain features (e.g., Fortran allocatable arrays, alternative returns, etc.) or the usage of contradicting philosophy (e.g., mutability or immutability of arguments in Fortran and Python, respectively, etc.). Solutions must be found to smoothen out such language differences that would not require transforming the original sources or would not contradict the philosophy of the target language.

This section gives an overview of how these issues can be resolved while striving to achieve the goals stated in the Section 1. Solutions to these problems are given and many are also applicable for wrapping Fortran procedures to other scripting languages. In particular, the solutions and design decisions under discussion are implemented in the F2PY tool (see Section 3).

2.1 Basic steps to create wrappers

Creating a Python wrapper to a Fortran procedure means creating a Python C/API extension module (called as wrapper module) that implements a Python extension function (written in C, called as wrapper function) which in turn calls the given Fortran procedure. In the wrapper function, before the actual call to Fortran, Python arguments must be converted to C data structures that are suitable for use as arguments to Fortran procedures. After the call, the wrapper function must convert the computed results to Python objects, and finally, return to the Python program. The wrapper module must be compiled and linked with the Fortran libraries to form a shared library that can be imported into Python.

Each of these steps requires some technical knowledge. To produce portable, robust, and easy to use wrappers, one needs to have experience with each of the various issues. It turns out that the greatest difficulties are due to variability of mixed-programming techniques within different Fortran compilers. A tool that automates the creation of wrappers, must implement algorithms that are independent of a particular choice of Fortran compiler. This task is not always as straightforward as one would wish, as we shall see in Section 2.4.

2.2 Collecting signature information from Fortran files

The signature of a Fortran procedure represents a complete set of information needed to call a Fortran procedure from a C program. Collecting the signatures from Fortran files is an important step towards automating interface generation because this interface information determines how the wrapper functions will convert Python argument objects to C data structures, how they will call Fortran procedures, and how they need to convert the results back to Python objects.

Fortran sources contain much information that is relevant to signatures. There are a number of possible approaches for implementing a program to extract signature information from Fortran source codes. The most obvious approach is to implement a Fortran parser or reuse some existing parser provided by a Fortran compiler. However, this approach has the disadvantage of requiring a huge amount of effort: implementing a full Fortran parser from scratch is a big project in itself, and reusing an existing Fortran compiler parser can be technically complicated because Fortran compilers usually do not provide interfaces to their parsers or they are hard to use for other applications such as wrapper generation tools that might need to extend the syntax.

An alternative and simpler approach is to implement a program that would scan Fortran source codes for signature patterns. Such a scanner should pick up only those Fortran statements that define the signatures of Fortran procedures. For example, only those type declaration statements need to be parsed that declare the types and dimensions of Fortran procedure arguments.

In general, the conversion methods of arguments in the wrapper function need to know about how different Fortran types may be represented as C objects and the size and shape of Fortran procedure arguments in the case of arrays. The size and shape information of array arguments are usually present as Fortran dimension specifications in a Fortran procedure. The basic Fortran types such as `INTEGER`, `REAL`, etc. can be immediately mapped to the corresponding C types `int`, `float`, etc., respectively. Of course, such a mapping must take into account the actual byte sizes of Fortran types that must match the byte sizes of the corresponding C types.

Note that, compared to Fortran subprograms, C programs contain typically less of the signature information required to create wrappers automatically (for example, arrays in C are declared as simple pointers without dimension information). This explains why tools designed to wrap C/C++ libraries to scripting languages need additional specification files describing the signatures, and why the task of wrapping C programs to scripting languages cannot be fully automated at the level that is possible with Fortran subprograms.

2.3 Ways to enhance the signature information

The choice of programming idioms used in a computer program depends strongly on the programming language. For example, in a typical Fortran 77 procedure computed results are often returned by updating the content of subroutine arguments (i.e., arguments are passed by reference or as addresses):

```
      subroutine foo (i1, i2, n1, n2, io3, o4,\
                      m1, m2, o5)
      integer n1, n2, m1, m2
      real i1, i2(n1, n2), io3
      real o4(m1, m2), o5(m1)
c Input arguments: i1, i2
c Input-output arguments: io3
c Output arguments: o4, o5
     ...
     end
```

In Python, however, changing arguments in-place is considered as poor Python programming style where output data from a function would, preferably, be returned as (multiple) objects. Note also that dimension information is included within array objects and, therefore, does not need to be passed via the argument list:

```
def foo(i1, i2, io3):
    ...
    return io3, o3, o4, o5
```

The above code examples represent so-called *function signatures* of Fortran and Python programs, respectively, in a notation used throughout this paper.

In general, Fortran procedures contain sufficient signature information to automatically generate wrappers. However, situations exist where the signature information needs to be changed by either adding more signature information or fixing the scanned signature information.

The most typical case for enhancing signature information comes when wrapping Fortran 77 procedures as these contain no information about the intention of arguments: whether the arguments contain just input data, or just results, or both. Such information can be very difficult or impossible to determine automatically without full program analysis while the user can often easily provide this additional information from documentation notes.

An example of where fixing signature information might be necessary is when wrapping legacy Fortran 77 code which abuses certain features of the Fortran 77 standards and compilers. An example is when array arguments of a Fortran procedure are declared to have size one but the callers may provide arrays with different sizes to the procedure (the actual sizes are specified in other arguments to the procedure). Obviously, no automatic tool can detect such abuses and automatically fix them.

One of the most common enhancements is to add intent information to argument declarations. This may be achieved either by

- Inserting the information via specially formatted comment in the Fortran source that can be detected by the scanner.
- Saving the scanned information to an intermediate specification file where the information can be modified by user. This modified file can be used to create the wrapper. The given approach is termed semi-automatic.

The syntax of both the specially formatted comment lines and intermediate specification files should follow the syntax of recent Fortran standards to reduce the learning curve for users with existing Fortran knowledge. Ideally, the scanner should be able to detect the errors when editing either the Fortran sources or the specification files.

2.4 Mixed language programming using Fortran and C

Technically, there are no difficulties in generating extension modules from a set of well-defined signatures. The generator must ensure that reference counting of Python objects is done correctly and that the generated code follows C standards to guarantee portability.

The difficult part of generating extension modules lies in mixed language programming between C and Fortran because the techniques for accessing Fortran symbols from C in a portable way, can be complex.

The internet and some Fortran compiler vendors provide technical documentations about mixed language programming techniques between C/C++ and Fortran 77/90/95, but there exists no standard specification about the techniques that all Fortran compilers would follow (except compilers that follow standard Fortran 2003). However, with at least one exception (the F compiler), all Fortran compilers (see Section 2.5)

have support for compiling fixed form Fortran sources containing standard Fortran 77 code. Moreover, the basic methods of mixed language programming techniques between Fortran 77 and C seem to be universal among all Fortran compilers assuming that only Fortran subroutines are used – this will be our assumption when discussing portable techniques. Recall that Fortran 77 is a subset of newer Fortran standards and the assumption does not put applicability restrictions to wrapping newer Fortran codes.

First, one needs to know how to access Fortran symbols from C. Different Fortran compilers map the names of Fortran symbols (the names of procedures, common blocks, modules, module data, module procedures, etc.) to the corresponding object names differently (name mangling). However, most Fortran compilers provide command line switches that can be used to achieve a particular naming convention. Such a convention can be used consistently in the wrapper generator. So, in the following we assume that when proper compiler switches are used, the name of a Fortran symbol (recall that names in Fortran are case-insensitive), say `FOO`, can be accessed from a C program as `foo_`, that is, all cases are lowered and exactly one underscore is appended to the name of a Fortran symbol. Note that sometimes the Fortran libraries provided by a system, may have been built using different compiler switches leading to different naming convention. For such cases, the wrapper generation tool must provide hooks to adapt default naming conventions to a particular case.

Unfortunately, the naming convention introduced above cannot be assumed for all Fortran 90 symbols. For example, the MIPSPro 7 compilers generate object code where Fortran 90 module names contain the $ character. Such names cannot be accessed from C directly and these compilers do not provide compile switches to change that behaviour. Therefore, more complex techniques are required to access Fortran 90 module symbols from C.

We propose the following original solution to the above problem: the references to Fortran 90 module symbols can be determined at runtime by using special wrappers. Basically, the initialisation function `initfoo` (see the example below) of a Python extension module `foo` calls an auxiliary Fortran subroutine `finitbar` in fixed form with a special call-back C function `init_bar` as an argument. The Fortran subroutine `finitbar` that uses Fortran 90 module `baz` symbols, makes a call-back to C using the call-back C function `init_bar`. The call-back function has the Fortran 90 symbols as arguments. In C, the pointers to these arguments are saved in global C variables that are visible to all C functions that need to access Fortran 90 symbols. This arrangement is best illustrated with the following example (the code should be read from the end for better understanding of the program flow):

```
! Fortran 90 code
module baz
  ! the aim is to reference bar
  ! from C (and Python)
  subroutine bar()
  end subroutine bar
end module baz

! auxiliary fixed format Fortran code
      subroutine finitbar(cinit)
      use baz
      extern cinit
      ! sending bar reference to C:
      call cinit(bar)
      end
/* Python C/API code */
extern void finitbar_;    /* GCC convention,
                             refers to Fortran
                             finitbar */
char *bar_ptr;            /* will hold reference
                             to bar */
void init_bar(char *bar) { /* initializes
                              bar_ptr */
 *bar_ptr = bar;
} static PyObject* bar_capi  /* wrapper function
                              to bar */
(PyObject *self, PyObject*args) {
 (*((void *)bar_ptr))();
} void initfoo() {     /* initializes
                 extension module */
 finitbar_(init_bar); /* initialize bar_ptr */
 ...
}
```

The proposed technique is unique because it makes mixing Fortran 90 and C as portable as mixing Fortran 77 and C codes. The technique can be applied also for referencing Fortran common blocks and Fortran 90 module data from C.

There exist other compiler related issues that the extension generation tool must take into account:

- Calling convention of `COMPLEX` or `CHARACTER` string valued Fortran functions is Fortran compiler dependent. The issue can be resolved by generating the auxiliary wrapper Fortran subroutine that executes the Fortran function call and saves the result to extra argument of the wrapper subroutine.
- Accessing and manipulating Fortran 90 module data `ALLOCATABLE` arrays requires additional methods that will support creating, initialising, modifying, deallocating, and reallocating such arrays. These operations are better executed in Fortran code than in C because often Fortran compilers do not provide any information how to access allocatable arrays as C structures (Pletzer et al., 2008).
- Dealing with Fortran alternative returns that have no matching paradigm in C.
- Handling Fortran `ENTRY` statements should produce additional wrapper functions for each such statement using the signature of the main subroutine.

2.5 Compiling and linking extension modules

In general, the task of compiling and linking the Python extension modules is best performed using Python's standard `distutils` package that implements extension module building support to as many platforms as Python itself supports. However, to build extension modules that use Fortran software, we recommended using the `numpy.distutils` package (provided by NumPy), which extends the standard `distutils` with support for Fortran compilers.

The main advantage of using `numpy.distutils` compared to the manual approach of building extension modules, is that `numpy.distutils` contains support to a large number of Fortran compilers. In addition, information about specific compilers (switches for universal naming conventions, compilation flags for optimised builds, required libraries for linking, etc.) are being continually updated when new Fortran compiler versions are released. At present, `numpy.distutils` supports the following list of Fortran compilers: GNU Fortran 77, GNU Fortran 95, Absoft Corp Fortran, Compaq Fortran, G95 Fortran, Intel/Intel Visual Fortran, Lahey/Fujitsu Fortran 95, NAGWare Fortran 95, Portland Group Fortran, Pacific-Sierra Research Fortran 90, HP Fortran 90, IBM XL Fortran, Sun or Forte Fortran 95, and MIPSpro Fortran compilers.

There may exist cases where building extension modules manually is still needed, for example, when one needs to have better control over how the extension modules and Fortran sources are compiled and linked together. For improving the portability of such setups, we recommend using the Fortran linker for creating extension module shared libraries because this will ensure that all necessary Fortran libraries are included in the linking step. Using the system linker requires determining this information manually which can be tedious because the information may change even between different versions of the same Fortran compiler.

2.6 Providing new signature information to wrapper users

In general, the process of generating Python wrapper functions to Fortran procedures requires the signatures of the original Fortran procedures to be transformed to functions with different signatures when viewed from the Python side (see Section 2.3). These transformations may be introduced automatically by the wrapper generation tool or explicitly by the user in order to make the wrapper functions more natural to use for Python programmers. The wrapper generation tool must generate documentation (doc strings) that precisely describe how the Fortran input and output arguments are treated from the Python side. Typically, a user must first check the documentation string before entering code for calling wrapper functions because the Python signature may differ from the Fortran signature, see Section 3.1 for an example.

3 The F2PY tool for connecting Fortran and Python programs

This section describes how the problem of connecting Fortran and Python programs in a robust and portable way is solved using the polished tool F2PY (F2PY), that is currently available as a part of the NumPy package. In the following we assume that the following software is installed: NumPy 1.1 (or newer), the Python development package `distutils` (some systems do not provide it by default when installing Python), a compatible C compiler, and a Fortran compiler supported by `numpy.distutils`.

3.1 A simple example

To get acquainted with F2PY, let us consider the following simple Fortran 77 function for computing the dot product of two vectors:

```
c file: dot.f
      FUNCTION dot(n, x, y)
      INTEGER n, i
      DOUBLE PRECISION dot, x(n), y(n)
      dot = 0d0
      DO 10 i = 1, n
        dot = dot + x(i) * y(i)
  10  CONTINUE
      END
```

In order to call the Fortran function `dot` from Python, let us create an interface to it by using `f2py` (F2PY's front-end program). For that we issue the following command:

```
sh> f2py -m foo dot.f -c
```

where the option `-m foo` sets the name of the extension module and the option `-c` instructs `f2py` to build the extension module library that can be imported to Python (without the `-c` option `f2py` would only generate sources to the extension module).[2]

The following sample session demonstrates how to call the Fortran function `dot` from Python:

```
sh> python
>>> import foo
>>> result = foo.dot([1,2], [3, 4])
>>> print result
11.0
```

Note the difference between the signatures of the Fortran function `dot(n, x, y)` and the corresponding wrapper function `dot(x, y)`:

```
>>> print foo.dot.__doc__
dot - Function signature:
  dot = dot(x,y,[n])
Required arguments:
  x : input rank-1 array('d') with bounds (n)
  y : input rank-1 array('d') with bounds (n)
Optional arguments:
  n := len(x) input int
Return objects:
  dot : float
```

Clearly, the latter is more convenient for the Python user: the user does not need to provide a value for `n` because the wrapper function can determine the value automatically from the length of the input vectors `x` and `y`. This feature often reduces very long Fortran argument lists to much shorter Python wrapper function argument lists.

3.2 Strategies of using F2PY

F2PY is designed to wrap Fortran codes of various complexity written with different styles of programming. We have worked out the following three basic strategies of wrapping Fortran procedures to Python:

- *The quick way* – let F2PY to do all the work, that is:
  - scan signature information from Fortran sources
  - create interface signatures for Fortran procedures, modules, and data collections
  - generate wrapper module sources containing necessary wrapper functions
  - compile C and Fortran source files
  - and finally, build the Python wrapper module that can be immediately used for calling Fortran procedures from Python.
- *The smart way* – let the user specify the interface signature (its initial version can be generated by using F2PY for scanning Fortran sources) and then let F2PY handle the rest as listed above.
- *The quick and smart way* – let the user specify extra signature information in Fortran source codes and then let F2PY to do all the work as specified under point 1.

The default strategy in F2PY is the quick way. Examples of using the quick way strategy were given above; wrapping the LAPACK library in Section 1 and the `dot` function in Section 3.1. The advantage of using the default strategy is that wrappers can be created with a single command and it provides immediate access to the Fortran code from Python. The disadvantage is that the user must be aware of the possible need to deal with the differences of the Fortran and C data storage orders as well as how to prepare arrays in Python so that they are efficiently passed to Fortran and filled with computed results. Such issues typically merge because of the lack of argument intent information in Fortran 77 procedures; though this is not a problem with Fortran 90 or newer procedures. As a result, F2PY must use the most conservative assumption: all arguments are input-only arguments. For example, in our LAPACK example, the array `b` has to have the `float64` type (that corresponds to Fortran `DOUBLE PRECISION` type) and Fortran storage order. Otherwise, the interface would create a copy of the input argument `b` and use the copy as the corresponding argument to the Fortran function. Note that although the computed results are saved to this copy array, the copy is not returned to Python because of the default input-only assumption.

The smart way strategy gives maximum flexibility to enhance the F2PY wrapper generation process and is appropriate for creating high-quality wrappers to (legacy) Fortran libraries. Typically, one uses the following iterative scheme for producing easily usable and efficient wrapper modules:

1 Create an initial wrapper module interface:

```
sh> f2py -m <modulename> <Fortran files>\
        -h <modulename>.pyf
```

This command will scan Fortran source files for procedure signatures and save the obtained signature information to a signature (`.pyf`) file.

2 Review the signature file manually: insert arguments intent information via the `intent` attribute, provide default values to optional arguments, etc.

3 Build the wrapper module:

```
sh> f2py -c <modulename>.pyf <Fortran files>
```

4 Check the documentation strings of the generated wrapper functions and test their functionality. In case anything needs to be enhanced, go back to Step 2.

The quick and smart way strategy is a combination of the two strategies described above. Here the additional signature information can be inserted directly to Fortran source codes using F2PY directive comments starting with `!f2py` that F2PY will process. As a result, one can create high-quality wrapper modules using just one F2PY command. This approach is suitable when one can add comments to Fortran source codes and has the advantage of keeping all signature information in one place.

3.3 Differences between Fortran and Python signatures

When F2PY generates a wrapper to a Fortran procedure then by default the resulting wrapper function follow Python conventions to make their usage easier for Python users. This behaviour of F2PY may change the interface of a Fortran procedure when viewed from the Python side, and users of F2PY must be aware of this (see Section 3.1). Although this is unfortunate from an interface stability point of view, when one takes into account the habits of typical Python users who are not familiar with Fortran idioms, being Pythonic outweighs possible disadvantages.

By default, F2PY automatically changes the signatures of Fortran procedures in the following two cases:

- When a Fortran procedure argument has an `intent(out)` attribute then the wrapper function returns the value of the given argument. The argument is also removed from the wrapper function argument list (in the case of not specifying an `intent(in)` attribute). For example, the following Fortran 77 subroutine,

```
subroutine foo(x, y)
!f2py intent(out) x, y
!f2py intent(in) y
...
```

  will have the following signature for the corresponding Python wrapper function:

```
def foo(y):
    ...
    return x, y
```

  Here we have used F2PY directives to illustrate how to insert argument intent information in Fortran 77 source code files.

- When an input array argument to a Fortran procedure has its dimension specified as an argument, then the dimension argument is made optional (and moved to the end of argument list) and its default value is set by the wrapper function using the shape information stored in the corresponding Python array argument. See Section 3.1 for an example.

3.4 Overview of F2PY features

In general, F2PY can be used to wrap *any* Fortran 77 and *certain* Fortran 90 or newer libraries to Python (see F2PY limitations below). The generated wrapper modules implement various consistency checks for wrapper functions to avoid program crashes and to give useful error messages when checks fail. Using F2PY auto-generated wrapper functions is made easy especially for Python users who have no prior knowledge of Fortran.

With some help from the developer of a wrapper module interface, the robustness and quality of F2PY generated wrapper functions can be easily improved by providing more information (than normally available from Fortran source codes) about the intentions of Fortran procedure arguments.

F2PY generated wrapper modules are highly portable and compiler independent. Currently F2PY has support for more than ten major Fortran compilers. F2PY implements the approaches explained in Section 2.4 to circumvent issues with compiler dependencies.

Although F2PY was originally designed to simplify wrapping large Fortran libraries to Python, the tool can be also used for wrapping certain C libraries to Python. The following rule of thumb can be used to decide if F2PY is suitable for wrapping a given C function to Python: if the C function is using Fortran 77 compatible argument types (e.g., not types derived from C `struct`) then F2PY is suitable for the task. Note that wrapping a C library cannot be made as automatic as wrapping a Fortran library because the relation between an array argument and its (integer) dimension arguments is not specified by the C code and the array size information must be provided manually. Many may feel that wrapping C functions with F2PY is significantly simpler than using other tools like SWIG. See http://www.scipy.org/Cookbook/f2py_and_NumPy for an example.

How F2PY interfaces various Fortran features to Python, are exemplified in the following sections. A detailed list of F2PY features can be found on the F2PY homepage and in the documentation.

3.4.1 Array arguments

As mentioned before, Fortran uses a different array storage order than C. This means that before passing Python multi-dimensional arrays (NumPy array objects) to Fortran procedures, the arrays must be transposed. In addition to that, when the Python array element type differs from that expected by a Fortran procedure (for example, when passing Python integer array to a Fortran real array argument) the array content must be copied to a new array with the expected element type. F2PY automatically performs the transposing and copying operations in one transformation and only then when really needed. Specifically, it is possible to create Python arrays with proper type and Fortran storage order that can be safely passed to Fortran without copying and transposing. F2PY can detect such array arguments and skip the corresponding transformations. This feature is very important because copying results in noticeable overheads, especially when dealing with large arrays and when calling Fortran procedures in a long Python loop. User can enable conditional reporting features when array copies are made (see `f2py` flag `-DF2PY_REPORT_ON_ARRAY_COPY`).

3.4.2 External procedures

Fortran procedures may declare `EXTERNAL` symbols which are procedures defined elsewhere. These external procedures can be specified as arguments to a Fortran procedure or as an object code to be linked with the Fortran program. F2PY supports both ways of specifying external procedures which makes it possible to call Python functions from Fortran procedures.

For example, when an external procedure is given as an argument to a Fortran procedure:

```
subroutine foo(bar)
external bar
call bar(...)
...
```

then F2PY automatically creates an interface that allows Python functions to be passed to Fortran::

```
>>> def bar(...):
        ...
>>> foo(bar)
```

where the Python function `bar` will be called with given arguments from the Fortran subroutine `bar` via an F2PY generated wrapper function `bar`. The problem of determining the signature of Fortran subroutine `fun` is solved by scanning Fortran `CALL` statements.

When the external procedure is not specified in a Fortran procedure argument list, for example:

```
subroutine foo()
external bar call
bar(...)
```

then the F2PY user must specify the `intent (callback)` attribute for the `bar` symbol that will instruct F2PY to create the necessary interface and add an additional argument to the wrapper function.

3.4.3 Common blocks

F2PY automatically generates interfaces to `COMMON` blocks defined in Fortran procedures and `DATA BLOCK` items. For example, the interface to a common block in a Fortran subroutine,

```
subroutine foo()
integer i
real x
common /data/ i, x(2, 3)
...
```

is exposed to Python via a special object whose attributes contain Python arrays with data referring to the corresponding Fortran common block content:

```
>>> data.i = 5 # sets the value of common \
                 block item i to 5
>>> data.x    # return the item x as \
                 writable array
```

Viewing or changing the content of `data` attributes in Python is directly related to viewing and changing the content of Fortran common block `data`.

3.4.4 Fortran 90 module procedures

F2PY automatically generates interfaces to Fortran 90 module procedures (Fortran 90 module subroutines and functions) and Fortran 90 module data. The interface is exposed via a special object that is created for each Fortran 90 module. In addition, F2PY generates interfaces to Fortran 90 allocatable arrays that are able to allocate, initialise, reallocate and deallocate the Fortran allocatable arrays from Python. For example, with the following Fortran 90 module:

```
module m
  real, allocatable, dimension(:) :: x
  contains
   subroutine foo
     ...
  ...
```

accessing the allocatable array `x` and the subroutine `foo` in Python looks trivial:

```
>>> m.foo()  # call F90 module m function foo
>>> m.x = [1, 2, 3] # allocate F90 x and \
                      initialize with \
                      given values
>>> m.x        # returns the content of \
                 x as array
>>> m.x = None # deallocate F90 x array
```

Here the F2PY generated interface handles the complicated job of accessing Fortran 90 symbols from C using the method explained in Section 2.4. This example is a good illustration of the power of F2PY: the tool provides a very convenient and simple Python access to Fortran 90 while behind the scenes it uses very complicated but unavoidable technique to ensure portability.

3.5 Limitations

The most important limitation of F2PY is that it can wrap only a certain subset of Fortran 90 programs. For example, F2PY does not support wrapping Fortran procedures that use arguments defined by the Fortran 90 `TYPE` or `POINTER` constructs.

At the time of writing, F2PY considers Fortran arrays of character strings as arrays of characters. This causes some inconvenience when providing NumPy string arrays to F2PY generated wrapper functions. (This restriction is present because the previous array backend package Numeric did not support arrays of strings.)

The other limitations of F2PY are often due to limitations of C. For example, Fortran alternative returns do not have the corresponding concept in C and hence F2PY cannot support alternative returns in a consistent manner.

4 Conclusions and future work

This paper has provided an overview of issues that arise from connecting low- and high-level languages, with particular focus on Fortran and Python. The most difficult problems come from the fact that mixed language programming between Fortran and C was not standardised until Fortran 2003 was published, and one needs to use nontrivial programming techniques in order to connect the Fortran (starting from Fortran 77) and Python programs in a reasonably portable way. In addition, one must be able to deal with the differences of the data storage orders that Fortran and C use for multi-dimensional arrays.

Examples were given highlighting the use of the software tool, F2PY, that solves many of the difficult mixed language programming problems. Strategies were suggested for dealing with idiomatic language differences between Fortran and Python in an efficient and convenient way.

In summary, F2PY can be used to (semi-)automatically wrap Fortran 77 and Fortran 90 or newer programs to Python, with the above mentioned restrictions. In practice, F2PY fulfils all optimal software solution criteria stated in Section 1 while special cases exist where expert support is needed. F2PY has become an important tool to speed up Python by migrating computationally intensive tasks to Fortran. F2PY has also given a new life to many legacy Fortran libraries and programs.

There is work in progress to extend F2PY to wrap a wider range of the Fortran 90, 95, and 2003 standard features, and most importantly add the Fortran 90 `TYPE` support. This may provide a significant impact on Computational Science, as scientists are able to reuse more contemporary high-performance Fortran subprograms in high-level Python programs.

Acknowledgements

Financial support from the Estonian Science Foundation is acknowledged (ETF grant 5767).

This work is supported by a Center of Excellence grant from the Norwegian Research Council to Center for Biomedical Computing at Simula Research Laboratory (http://www.simula.no/).

Support to develop F2PY from the Centre of Nonlinear Studies (http://cens.ioc.ee/) and Enthought Inc. (http://www.enthought.com/) is greatly appreciated.

I am grateful to all F2PY users for providing feedback and bug reports on F2PY. Special thanks are due to Travis E. Oliphant who was the first F2PY user and contributor.

References

Beazley, D.M. (2003) 'SWIG: an extensible compiler for creating scriptable scientific software', *Future Generation Computer Systems (FGCS)*, Elsevier, Vol. 19, No. 5, pp.599–609, DOI: 10.1016/S0167-739X(02)00171-1.

Oliphant, T.E. (2007) 'Python for scientific computing', *Computing in Science and Engineering*, Vol. 9, No. 3, May–June, pp.10–20.

Ousterhout, J.K. (1998) 'Scripting: higher level programming for the 21st century', *Computer IEEE*, Vol. 31, No. 3, March, pp.23–30.

Peterson, P., Martins, J.R.R.A. and Alonso, J.J. (2001) 'Fortran to Python interface generator with an application to aerospace engineering', *Proceedings of the 9th International Python Conference*, CDROM, Long Beach, California, 19 pages.

Prechelt, L. (2000) 'An empirical comparison of seven programming languages', *Computer IEEE*, Vol. 33, No. 10, pp.23–29.

Pletzer, A., McCune, D., Muszala, S., Vadlamani, S. and Kruger, S. (2008) 'Exposing Fortran derived types to C and other languages', *Computing in Science and Engineering*, Vol. 10, No. 4, July–August, pp.86–92.

Wilson, G.V. (2006) 'Where's the real bottleneck in scientific computing', *American Scientist*, Sigma Xi, Vol. 94, No. 1, pp.5, 6, DOI: 10.1511/2006.1.5.

Notes

[1] A Pythonic interface means that it should be easy and natural to use for Python users, i.e., it should support coding styles and code idioms that have been well established by the Python community.

[2] To learn more about `f2py` command line options, run `f2py` without arguments.

Website

F2PY: Fortran to Python interface generator, http://www.f2py.org/