Fortran to Python Interface Generator with an Application to Aerospace Engineering

Pearu Peterson
<pearu@cens.ioc.ee>
Center of Nonlinear Studies
Institute of Cybernetics at TTU
Akadeemia Rd 21, 12618 Tallinn, ESTONIA

Joaquim R. R. A. Martins and Juan J. Alonso
<joaquim.martins@stanford.edu>, <jjalonso@stanford.edu>
Department of Aeronautics and Astronautics
Stanford University, CA

Revision : 1.16
January 18, 2001

Abstract

FPIG — Fortran to Python Interface Generator — is a tool for generating Python C/API extension modules that interface Fortran 77/90/95 codes with Python. This tool automates the process of interface generation by scanning the Fortran source code to determine the signatures of Fortran routines and creating a Python C/API module that contains the corresponding interface functions. FPIG also attempts to find dependence relations between the arguments of a Fortran routine call (e.g. an array and its dimensions) and constructs interface functions with potentially fewer arguments. The tool is extremely flexible since the user has control over the generation process of the interface by specifying the desired function signatures. The home page for FPIG can be found at http://cens.ioc.ee/projects/f2py2e/.

FPIG has been used successfully to wrap a large number of Fortran programs and libraries. Advances in computational science have led to large improvements in the modeling of physical systems which are often a result of the coupling of a variety of physical models that were typically run in isolation. Since a majority of the available physical models have been previously written in Fortran, the importance of FPIG in accomplishing these couplings cannot be understated. In this paper, we present an application of FPIG to create an object-oriented framework for aero-structural analysis and design of aircraft.

Contents

1 Preface

The use of high-performance computing has made it possible to tackle many important problems and discover new physical phenomena in science and engineering. These accomplishments would not have been achieved without the computer's ability to process large amounts of data in a reasonably short time. It can safely be said that the computer has become an essential tool for scientists and engineers. However, the diversity of problems in science and engineering has left its mark as computer programs have been developed in different programming languages, including languages developed to describe certain specific classes of problems.

In interdisciplinary fields it is not uncommon for scientists and engineers to face problems that have already been solved in a different programming environment from the one they are familiar with. Unfortunately, researchers may not have the time or willingness to learn a new programming language and typically end up developing the corresponding tools in the language that they normally use. This approach to the development of new software can substantially impact the time to develop and the quality of the resulting product: firstly, it usually takes longer to develop and test a new tool than to learn a new programming environment, and secondly it is very unlikely that a non-specialist in a given field can produce a program that is more efficient than more established tools.

To avoid situations such as the one described above, one alternative would be to provide automatic or semi-automatic interfaces between programming languages. Another possibility would be to provide language translators, but these obviously require more work than interface generators — a translator must understand all language constructs while an interface generator only needs to understand a subset of these constructs. With an automatic interface between two languages, scientists or engineers can effectively use programs written in other programming languages without ever having to learn them.

Although it is clear that it is impossible to interface arbitrary programming languages with each other, there is no reason for doing so. Low-level languages such as C and Fortran are well known for their speed and are therefore suitable for applications where performance is critical. High-level scripting languages, on the other hand, are generally slower but much easier to learn and use, especially when performing interactive analysis. Therefore, it makes sense to create interfaces only in one direction: from lower-level languages to higher-level languages.

In an ideal world, scientists and engineers would use higher-level languages for the manipulation of the mathematical formulas in a problem rather than having to struggle with tedious programming details. For tasks that are computationally demanding, they would use interfaces to high-performance routines that are written in a lower-level language optimized for execution speed.

2 Introduction

This paper presents a tool that has been developed for the creation of interfaces between Fortran and Python.

The Fortran language is popular in scientific computing, and is used mostly in applications that use extensive matrix manipulations (e.g. linear algebra). Since Fortran has been the standard language among scientists and engineers for at least three decades, there is a large number of legacy codes available that perform a variety of tasks using very sophisticated algorithms (see e.g. [1]).

The Python language [2], on the other hand, is a relatively new programming language. It is a very high-level scripting language that supports object-oriented programming. What makes Python especially appealing is its very clear and natural syntax, which makes it easy to learn and use. With Python one can implement relatively complicated algorithms and tasks in a short time with very compact source code.

Although there are ongoing projects for extending Python's usage in scientific computation, it lacks reliable tools that are common in scientific and engineering such as ODE integrators, equation solvers, tools for FEM, etc. The implementation of all of these tools in Python would be not only too time-consuming but also inefficient. On the other hand, these tools are already developed in other, computationally more efficient languages such as Fortran or C. Therefore, the perfect role for Python in the context of scientific computing would be that of a "gluing" language. That is, the role of providing high-level interfaces to C, C++ and Fortran libraries.

There are a number of widely-used tools that can be used for interfacing software libraries to Python. For binding C libraries with various scripting languages, including Python, the tool most often used is SWIG [3]. Wrapping Fortran routines with Python is less popular, mainly because there are many platform and compiler-specific issues that need to be addressed. Nevertheless, there is great interest in interfacing Fortran libraries because they provide invaluable tools for scientific computing. At LLNL, for example, a tool called PyFort has been developed for connecting Fortran and Python [4].

The tools mentioned above require an input file describing signatures of functions to be interfaced. To create these input files, one needs to have a good knowledge of either C or Fortran. In addition, binding libraries that have thousands of routines can certainly constitute a very tedious task, even with these tools.

The tool that is introduced in this paper, FPIG (Fortran to Python Interface Generator) [5], automatically generates interfaces between Fortran and Python. It is different from the tools mentioned above in that FPIG can create signature files automatically by scanning the source code of the libraries and then construct Python C/API extension modules. Note that the user need not be experienced in C or even Fortran. In addition, FPIG is designed to wrap large Fortran libraries containing many routines with only one or two commands. This process is very flexible since one can always modify the generated signature files to insert additional attributes in order to achieve more sophisticated interface functions such as taking care of optional arguments, predicting the sizes of array arguments and performing various checks on the correctness of the input arguments.

The organization of this paper is as follows. First, a simple example of FPIG usage is given. Then FPIG's basic features are described and solutions to platform and compiler specific issues are discussed. Unsolved problems and future work on FPIG's development are also addressed. Finally, an application to a large aero-structural solver is presented as real-world example of FPIG's usage.

3 Getting Started

To get acquainted with FPIG, let us consider the simple Fortran 77 subroutine shown in Fig. 1. In the sections that follow, two ways of creating interfaces to this Fortran subroutine are described. The first and simplest way is suitable for Fortran codes that are developed in connection with `f2py`. The second and not much more difficult method, is suitable for interfacing existing Fortran libraries which might have been developed by other programmers.

Numerical Python [6] is needed in order to compile extension modules generated by FPIG.

3.1 Interfacing Simple Routines

In order to call the Fortran routine `exp1` from Python, let us create an interface to it by using `f2py` (FPIG's front-end program). In order to do this, we issue the following command,

```
      subroutine exp1(l,u,n)
C     Input: n is number of iterations
C     Output: l,u are such that
C       l(1)/l(2) < exp(1) < u(1)/u(2)
C
Cf2py integer*4 :: n = 1
Cf2py intent(out) l,u
      integer*4 n,i
      real*8 l(2),u(2),t,t1,t2,t3,t4
      l(2) = 1
      l(1) = 0
      u(2) = 0
      u(1) = 1
      do 10 i=0,n
         t1 = 4 + 32*(1+i)*i
         t2 = 11 + (40+32*i)*i
         t3 = 3 + (24+32*i)*i
         t4 = 8 + 32*(1+i)*i
         t = u(1)
         u(1) = l(1)*t1 + t*t2
         l(1) = l(1)*t3 + t*t4
         t = u(2)
         u(2) = l(2)*t1 + t*t2
         l(2) = l(2)*t3 + t*t4
 10   continue
      end
```

Figure 1: Example Fortran code `exp1.f`. This routine calculates the simplest rational lower and upper approximations to e (for details of the algorithm see [7], p.122)

```
sh> f2py -m foo exp1.f
```

where the option `-m foo` sets the name of the Python C/API extension module that `f2py` will create to `foo`. To learn more about the `f2py` command line options, run `f2py` without arguments.

The output messages in Fig. 2 illustrate the procedure followed by `f2py`: (i) it scans the Fortran source code specified in the command line, (ii) it analyses and determines the routine signatures, (iii) it constructs the corresponding Python C/API extension modules, (iv) it writes documentation to a LaTeX file, and (v) it creates a GNU Makefile for building the shared modules.

Now we can build the `foo` module:

```
sh> make -f Makefile-foo
```

Figure 3 illustrates a sample session for calling the

```
Reading fortran codes...
  Reading file 'exp1.f'
Post-processing...
  Block: foo
              Block: exp1
Creating 'Makefile-foo'...
  Linker: ld ('GNU ld' 2.9.5)
  Fortran compiler: f77 ('g77 2.x.x' 2.95.2)
  C compiler: cc ('gcc 2.x.x' 2.95.2)
Building modules...
  Building module "foo"...
      Constructing wrapper function "exp1"...
        l,u = exp1([n])
  Wrote C/API module "foo" to file "foomodule.c"
  Documentation is saved to file "foomodule.tex"
Run GNU make to build shared modules:
        gmake -f Makefile-<modulename> [test]
```

Figure 2: Output messages of `f2py -m foo exp1.f`.

```
>>> import foo,Numeric
>>> print foo.exp1.__doc__
exp1 - Function signature:
  l,u = exp1([n])
Optional arguments:
  n := 1 input int
Return objects:
  l : rank-1 array('d') with bounds (2)
  u : rank-1 array('d') with bounds (2)

>>> l,u = foo.exp1()
>>> print l,u
[ 1264.   465.] [ 1457.   536.]
>>> print l[0]/l[1], u[0]/u[1]-l[0]/l[1]
2.71827956989 2.25856657199e-06
>>> l,u = foo.exp1(2)
>>> print l,u
[ 517656.  190435.] [ 566827.  208524.]
>>> print l[0]/l[1], u[0]/u[1]-l[0]/l[1]
2.71828182845 1.36437527942e-11
```

Figure 3: Calling Fortran routine `exp1` from Python. Here `l[0]/l[1]` gives an estimate to e with absolute error less than `u[0]/u[1]-l[0]/l[1]` (this value may depend on the platform and compiler used).

Fortran routine `exp1` from Python.

Note the difference between the signatures of the Fortran routine `exp1(l,u,n)` and the corresponding wrapper function `l,u=exp1([n])`. Clearly, the later is more informative to the user: `exp1` takes one optional argument `n` and it returns `l`, `u`. This exchange of signatures is achieved by special comment lines (starting with `Cf2py`) in the Fortran source code — these lines are interpreted by `f2py` as normal Fortran code. Therefore, in the given example the line `Cf2py integer*4 :: n = 1` informs `f2py` that the variable `n` is optional with a default value equal to one. The line `Cf2py intent(out) l,u` informs `f2py` that the variables `l,u` are to be returned to Python after calling Fortran function `exp1`.

3.2 Interfacing Libraries

In our example the Fortran source `exp1.f` contains `f2py` specific information, though only as comments. When interfacing libraries from other parties, it is not recommended to modify their source. Instead, one should use a special auxiliary file to collect the signatures of all Fortran routines and insert `f2py` specific declaration and attribute statements in that file. This auxiliary file is called a *signature file* and is identified by the extension `.pyf`.

We can use `f2py` to generate these signature files by using the `-h <filename>.pyf` option. In our example, `f2py` could have been called as follows,

```
sh> f2py -m foo -h foo.pyf exp1.f
```

where the option `-h foo.pyf` requests `f2py` to read the routine signatures, save them to the file `foo.pyf`, and then exit. If `exp1.f` in Fig. 1 were to contain no lines starting with `Cf2py`, the corresponding signature file `foo.pyf` would be as shown in Fig. 4. In order to obtain the exchanged and more convenient signature `l,u=foo.exp1([n])`, we would edit `foo.pyf` as shown in Fig. 5. The Python C/API extension module `foo` can be constructed by applying `f2py` to the signature file with the following command:

```
sh> f2py foo.pyf
```

The procedure for building the corresponding shared module and using it in Python is identical to the one described in the previous section.

As we can see, the syntax of the signature file is an extension of the Fortran 90/95 syntax. This means that only a few new constructs are introduced for `f2py` in addition to all standard Fortran constructs; signature files can even be written in fixed form. A complete set of constructs that are used when creating interfaces, is described in the `f2py` User's Guide [8].

4 Basic Features

In this section a short overview of `f2py` features is given.

```
!%f90 -*- f90 -*-
python module foo
    interface
        subroutine exp1(l,u,n)
            real*8 dimension(2) :: l
            real*8 dimension(2) :: u
            integer*4 :: n
        end subroutine exp1
    end interface
end python module foo
! This file was auto-generated with f2py
! (version:2.298).
! See http://cens.ioc.ee/projects/f2py2e/
```

Figure 4: Raw signature file `foo.pyf` generated with `f2py -m foo -h foo.pyf exp1.f`

```
!%f90 -*- f90 -*-
python module foo
    interface
        subroutine exp1(l,u,n)
            real*8 dimension(2) :: l
            real*8 dimension(2) :: u
            intent(out) l,u
            integer*4 optional :: n = 1
        end subroutine exp1
    end interface
end python module foo
! This file was auto-generated with f2py
! (version:2.298) and modified by pearu.
! See http://cens.ioc.ee/projects/f2py2e/
```

Figure 5: Modified signature file `foo.pyf`

1. All basic Fortran types are supported. They include the following type specifications:

   ```
   integer[ | *1 | *2 | *4 | *8 ]
   logical[ | *1 | *2 | *4 | *8 ]
   real[ | *4 | *8 | *16 ]
   complex[ | *8 | *16 | *32 ]
   double precision, double complex
   character[ |*(*)|*1|*2|*3|...]
   ```

   In addition, they can all be in the kind-selector form (e.g. `real(kind=8)`) or char-selector form (e.g. `character(len=5)`).

2. Arrays of all basic types are supported. Dimension specifications can be of form `<dimension>` or `<start>:<end>`. In addition, `*` and `:` dimension specifications can be used for input arrays. Dimension specifications may contain also `PARAMETER`'s.

3. The following attributes are supported:

   - `intent(in)`: used for input-only arguments.
   - `intent(inout)`: used for arguments that are changed in place.
   - `intent(out)`: used for return arguments.
   - `intent(hide)`: used for arguments to be removed from the signature of the Python function.
   - `intent(in,out)`, `intent(inout,out)`: used for arguments with combined behavior.
   - `dimension(<dimspec>)`
   - `depend([<names>])`: used for arguments that depend on other arguments in `<names>`.
   - `check([<C booleanexpr>])`: used for checking the correctness of input arguments.
   - `note(<LaTeX text>)`: used for adding notes to the module documentation.
   - `optional`, `required`
   - `external`: used for call-back arguments.
   - `allocatable`: used for Fortran 90/95 allocatable arrays.

4. Using `f2py` one can call arbitrary Fortran 77/90/95 subroutines and functions from Python, including Fortran 90/95 module routines.

5. Using `f2py` one can access data in Fortran 77 COMMON blocks and variables in Fortran 90/95 modules, including allocatable arrays.

6. Using `f2py` one can call Python functions from Fortran (call-back functions). `f2py` supports very flexible hooks for call-back functions.

7. Wrapper functions perform the necessary type conversations for their arguments resulting in contiguous Numeric arrays that are suitable for passing to Fortran routines.

8. `f2py` generates documentation strings for `__doc__` attributes of the wrapper functions automatically.

9. `f2py` scans Fortran codes and creates the signature files. It automatically detects the signatures of call-back functions, solves argument dependencies, decides the order of initialization of optional arguments, etc.

10. `f2py` automatically generates GNU Makefiles for compiling Fortran and C codes, and linking them to a shared module. `f2py` detects available Fortran and C compilers. The supported compilers include the GNU project C Compiler (gcc), Compaq Fortran, VAST/f90 Fortran, Absoft F77/F90, and MIPSpro 7 Compilers, etc. `f2py` has been tested to work on the following platforms: Intel/Alpha Linux, HP-UX, IRIX64.

11. Finally, the complete `f2py` User's Guide is available in various formats (ps, pdf, html, dvi). A mailing list, `<f2py-users@cens.ioc.ee>`, is open for support and feedback. See the FPIG's home page for more information [5].

5 Implementation Issues

The Fortran to Python interface can be thought of as a three layer "sandwich" of different languages: Python, C, and Fortran. This arrangement has two interfaces: Python-C and C-Fortran. Since Python itself is written in C, there are no basic difficulties in implementing the Python-C interface [9]. The C-Fortran interface, on the other hand, results in many platform and compiler specific issues that have to be dealt with. We will now discuss these issues in some detail and describe how they are solved in FPIG.

5.1 Mapping Fortran Types to C Types

Table 1 defines how Fortran types are mapped to C types in `f2py`. Users may redefine these mappings by creating a `.f2py_f2cmap` file in the working directory. This file should contain a Python dictionary of dictionaries, e.g. `{'real':{'low':'float'}}`, that informs `f2py` to map Fortran type `real(low)` to C type `float` (here `PARAMETER low = ...`).

5.2 Calling Fortran (Module) Routines

When mixing Fortran and C codes, one has to know how function names are mapped to low-level symbols in their object files. Different compilers may use different conventions for this purpose. For example, gcc appends the underscore _ to a Fortran routine name. Other compilers may use upper case names, prepend or append different symbols to Fortran routine names or both. In any case, if the low-level symbols corresponding to Fortran routines are valid for the C language specification, compiler specific issues can be solved by using CPP macro features.

Fortran type	C type
`integer *1`	`char`
`byte`	`char`
`integer *2`	`short`
`integer[ | *4]`	`int`
`integer *8`	`long long`
`logical *1`	`char`
`logical *2`	`short`
`logical[ | *4]`	`int`
`logical *8`	`int`
`real[ | *4]`	`float`
`real *8`	`double`
`real *16`	`long double`
`complex[ | *8]`	`struct {float r,i;}`
`complex *16`	`struct {double r,i;}`
`complex *32`	`struct {long double r,i;}`
`character[*...]`	`char *`

Table 1: Mapping Fortran types to C types.

Unfortunately, there are Fortran compilers that use symbols in constructing low-level routine names that are not valid for C. For example, the (IRIX64) MIPSpro 7 Compilers use '$' character in the low-level names of module routines which makes it impossible (at least directly) to call such routines from C when using the MIPSpro 7 C Compiler.

In order to overcome this difficulty, FPIG introduces an unique solution: instead of using low-level symbols for calling Fortran module routines from C, the references to such routines are determined at run-time by using special wrappers. These wrappers are called once during the initialization of an extension module. They are simple Fortran subroutines that use a Fortran module and call another C function with Fortran module routines as arguments in order to save their references to C global variables that are later used for calling the corresponding Fortran module routines. This arrangement is set up as follows. Consider the following Fortran 90 module with the subroutine `bar`:

```
module fun
  subroutine bar()
  end
end
```

Figure 6 illustrates a Python C/API extension module for accessing the F90 module subroutine `bar` from Python. When the Python module `foo` is loaded, `finitbar` is called. `finitbar` calls `init_bar` by passing the reference of the Fortran 90 module subroutine `bar` to C where it is saved to the variable `bar_ptr`. Now, when one executes `foo.bar()` from Python, bar_ptr is used in bar_capi to call the F90 module subroutine bar.

```
#include "Python.h"
...
char *bar_ptr;
void init_bar(char *bar) {
  bar_ptr = bar;
}
static PyObject *
bar_capi(PyObject *self,PyObject *args) {
  ...
  (*((void *)bar_ptr))();
  ...
}
static PyMethodDef
foo_module_methods[] = {
  {"bar",bar_capi,METH_VARARGS},
  {NULL,NULL}
};
extern void finitbar_; /* GCC convention */
void initfoo() {
  ...
  finitbar_(init_bar);
  Py_InitModule("foo",foo_module_methods);
  ...
}
```

Figure 6: Sketch of Python C/API for accessing F90 module subroutine bar. The Fortran function finitbar is defined in Fig. 7.

```
      subroutine finitbar(cinit)
        use fun
        extern cinit
        call cinit(bar)
      end
```

Figure 7: Wrapper for passing the reference of bar to C code.

Surprisingly, mixing C code and Fortran modules in this way is as portable and compiler independent as mixing C and ordinary Fortran 77 code.

Note that extension modules generated by f2py actually use PyFortranObject that implements above described scheme with exchanged functionalities (see Section 5.5).

5.3 Wrapping Fortran Functions

The Fortran language has two types of routines: subroutines and functions. When a Fortran function returns a composed type such as COMPLEX or CHARACTER-array then calling this function directly from C may not work for all compilers, as C functions are not supposed to return such references. In order to avoid this, FPIG constructs an additional Fortran wrapper subroutine for each such Fortran function. These wrappers call just the corresponding functions in the Fortran layer and return the result to C through its first argument.

5.4 Accessing Fortran Data

In Fortran one can use COMMON blocks and Fortran module variables to save data that is accessible from other routines. Using FPIG, one can also access these data containers from Python. To achieve this, FPIG uses special wrapper functions (similar to the ones used for wrapping Fortran module routines) to save the references to these data containers so that they can later be used from C.

FPIG can also handle allocatable arrays. For example, if a Fortran array is not yet allocated, then by assigning it in Python, the Fortran to Python interface will allocate and initialize the array. For example, the F90 module allocatable array bar defined in

```
module fun
  integer, allocatable :: bar(:)
end module
```

can be allocated from Python as follows

```
>>> import foo
>>> foo.fun.bar = [1,2,3,4]
```

5.5 PyFortranObject

In general, we would like to access from Python the following Fortran objects:

- subroutines and functions,
- F90 module subroutines and functions,
- items in COMMON blocks,
- F90 module data.

Assuming that the Fortran source is available, we can determine the signatures of these objects (the full specification of routine arguments, the layout of Fortran data, etc.). In fact, f2py gets this information while scanning the Fortran source.

In order to access these Fortran objects from C, we need to determine their references. Note that the direct access of F90 module objects is extremely compiler dependent and in some cases even impossible.

Therefore, FPIG uses various wrapper functions for obtaining the references to Fortran objects. These wrapper functions are ordinary F77 subroutines that can easily access objects from F90 modules and that pass the references to Fortran objects as C variables.

f2py generated Python C/API extension modules use PyFortranObject to store the references of Fortran objects. In addition to the storing functionality, the PyFortranObject also provides methods for accessing/calling Fortran objects from Python in a user-friendly manner. For example, the item a in COMMON /bar/ a(2) can be accessed from Python as foo.bar.a.

Detailed examples of PyFortranObject usage can be found in [10].

5.6 Callback Functions

Fortran routines may have arguments specified as external. These arguments are functions or subroutines names that the receiving Fortran routine will call from its body. For such arguments FPIG constructs a call-back mechanism (originally contributed by Travis Oliphant) that allows Fortran routines to call Python functions. This is actually realized using a C layer between Python and Fortran. Currently, the call-back mechanism is compiler independent unless a call-back function needs to return a composed type (e.g. COMPLEX).

The signatures of call-back functions are determined when f2py scans the Fortran source code. To illustrate this, consider the following example:

```
subroutine foo(bar, fun, boo)
  integer i
  real r
  external bar,fun,boo
  call bar(i, 1.2)
  r = fun()
  call sun(boo)
end
```

f2py recognizes the signatures of the user routines bar and fun using the information contained in the lines call bar(i, 1.2) and r = fun():

```
subroutine bar(a,b)
  integer a
  real b
end
function fun()
  real fun
end
```

But f2py cannot determine the signature of the user routine boo because the source contains no information at all about the boo specification. Here user needs to provide the signature of boo manually.

6 Future Work

FPIG can be used to wrap almost any Fortran code. However, there are still issues that need to be resolved. Some of them are listed below:

1. One of the FPIG's goals is to become as platform and compiler independent as possible. Currently FPIG can be used on any UN*X platform that has gcc installed in it. In the future, FPIG should be also tested on Windows systems.

2. Another goal of FPIG is to become as simple to use as possible. To achieve that, FPIG should start using the facilities of distutils, the new Python standard to distribute and build Python modules. Therefore, a contribution to distutils that can handle Fortran extensions should be developed.

3. Currently users must be aware of the fact that multi-dimensional arrays are stored differently in C and Fortran (they must provide transposed multi-dimensional arrays to wrapper functions). In the future a solution should be found such that users do not need to worry about this rather confusing and technical detail.

4. Finally, a repository of signature files for widely-used Fortran libraries (e.g. BLAS, LAPACK, MINPACK, ODEPACK, EISPACK, LINPACK) should be provided.

7 Application to a Large Aero-Structural Analysis Framework

7.1 The Need for Python and FPIG

As a demonstration of the power and usefulness of FPIG, we will present work that has been done at the Aerospace Computing Laboratory at Stanford University. The focus of the research is on aircraft design optimization using high-fidelity analysis tools such as Computational Fluid Dynamics (CFD) and Computational Structural Mechanics (CSM) [11].

The group's analysis programs are written mainly in Fortran and are the result of many years of development. Until now, any researcher that needed to use these tools would have to learn a less than user-friendly interface and become relatively familiar with the inner workings of the codes before starting the research itself. The need to couple analyses of different disciplines revealed the additional inconvenience of gluing and scripting the different codes with Fortran.

It was therefore decided that the existing tools should be wrapped using an object-oriented language in order to improve their ease of use and versatility. The use of several different languages such as C++, Java and Perl was investigated but Python seemed to provide the best solution. The fact that it combines scripting capability with a fully-featured object-oriented programming language, and that it has a clean syntax were factors that determined our choice. The introduction of tools that greatly facilitate the task of wrapping Fortran with Python provided the final piece needed to realize our objective.

7.2 Wrapping the Fortran Programs

In theory, it would have been possible to wrap our Fortran programs with C and then with Python by hand. However, this would have been a labor intensive task that would detract from our research. The use of tools that automate the task of wrapping has been extremely useful.

The first such tool that we used was PyFort. This tool created the C wrappers and Python modules automatically, based on signature files (`.pyf`) provided by the user. Although it made the task of wrapping considerably easier, PyFort was limited by the fact that any Fortran data that was needed at the Python level had to be passed in the argument list of the Fortran subroutine. Since the bulk of the data in our programs is shared by using Fortran 77 common blocks and Fortran 90 modules, this required adding many more arguments to the subroutine headers. Furthermore, since Fortran does not allow common block variables or module data to be specified in a subroutine argument list, a dummy pointer for each desired variable had to be created and initialized.

The search for a better solution to this problem led us to `f2py`. Since `f2py` provides a solution for accessing common block and module variables, there was no need to change the Fortran source anymore, making the wrapping process even easier. With `f2py` we also experienced an increased level of automation since it produces the signature files automatically, as well as a Makefile for the joint compilation of the original Fortran and C wrapper codes. This increased automation did not detract from its flexibility since it was always possible to edit the signature files to provide different functionality.

Once Python interfaces were created for each Fortran application by running `f2py`, it was just a matter of using Python to achieve the final objective of developing an object-oriented framework for our multidisciplinary solvers. The Python modules that we designed are discussed in the following section.

7.3 Module Design

The first objective of this effort was to design the classes for each type of analysis, each representing an independent Python module. In our case, we are interested in performing aero-structural analysis and optimization of aircraft wings. We therefore needed an analysis tool for the flow (CFD), another for analyzing the structure (CSM), as well as a geometry database. In addition, we needed to interface these two tools in order to analyze the coupled system. The object design for each of these modules should be general enough that the underlying analysis code in Fortran can be changed without changing the Python interface. Another requirement was that the modules be usable on their own for single discipline analysis.

7.3.1 Geometry

The *Geometry* class provides a database for the outer mold geometry of the aircraft. This database needs to be accessed by both the flow and structural solvers. It contains a parametric description of the aircraft's surface as well as methods that extract and update this information.

7.3.2 Flow

The flow solver was wrapped in a class called *Flow*. The class was designed so that it can wrap any type of CFD solver. It contains two main objects: the computational mesh and a solver object. A graph showing the hierarchy of the objects in *Flow* is shown in Fig. 8. Methods in the flow class include those used for the initialization of all the class components as well as methods that write the current solution to a file.

7.3.3 Structure

The *Structure* class wraps a structural analysis code. The class stores the information about the structure itself in an object called *Model* which also provides methods for changing and exporting its information. A list of the objects contained in this class can be seen in Fig. 9. Since the *Structure* class contains a dictionary of *LoadCase* objects, it is able to store

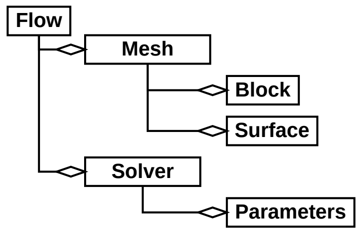

Figure 8: The *Flow* container class.

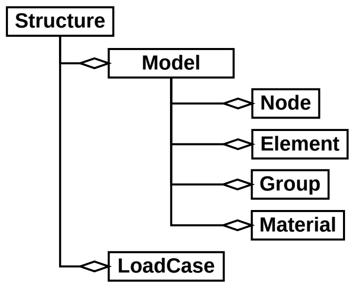

Figure 9: The *Structure* container class.

and solve multiple load cases, a capability that the original Fortran code does not have.

7.3.4 Aerostructure

The *Aerostructure* class is the main class in the aerostructural analysis module and contains a *Geometry*, a *Flow* and a *Structure*. In addition, the class defines all the functions that are necessary to translate aerodynamic loads to structural loads and structural displacements to geometry surface deformations.

One of the main methods of this class is the one that solves the aeroelastic system. This method is printed below:

```
def Iterate(self, load_case):
  """Iterates the aero-structural solution."""
  self.flow.Iterate()
  self._UpdateStructuralLoads()
  self.structure.CalcDisplacements(load_case)
  self.structure.CalcStresses(load_case)
  self._UpdateFlowMesh()
  return
```

This is indeed a very readable script, thanks to Python, and any high-level changes to the solution procedure can be easily implemented. The *Aerostructure* class also contains methods that export all the information on the current solution for visualization, an example of which is shown in the next section.

7.4 Results

In order to visualize results, and because we needed to view results from multiple disciplines simultaneously, we selected OpenDX. Output files in DX format are written at the Python level and the result can be seen in Fig. 10 for the case of a transonic airliner configuration.

The figure illustrates the multidisciplinary nature of the problem. The grid pictured in the background is the mesh used by the flow solver and is colored by the pressure values computed at the cell centers. The wing in the foreground and its outer surface is clipped to show the internal structural components which are colored by their stress value.

In conclusion, `f2py` and Python have been extremely useful tools in our pursuit for increasing the usability and flexibility of existing Fortran tools.

References

[1] Netlib repository at UTK and ORNL.
http://www.netlib.org/

[2] Python language.
http://www.python.org/

[3] SWIG — Simplified Wrapper and Interface Generator.
http://www.swig.org/

[4] PyFort — The Python-Fortran connection tool.
http://pyfortran.sourceforge.net/

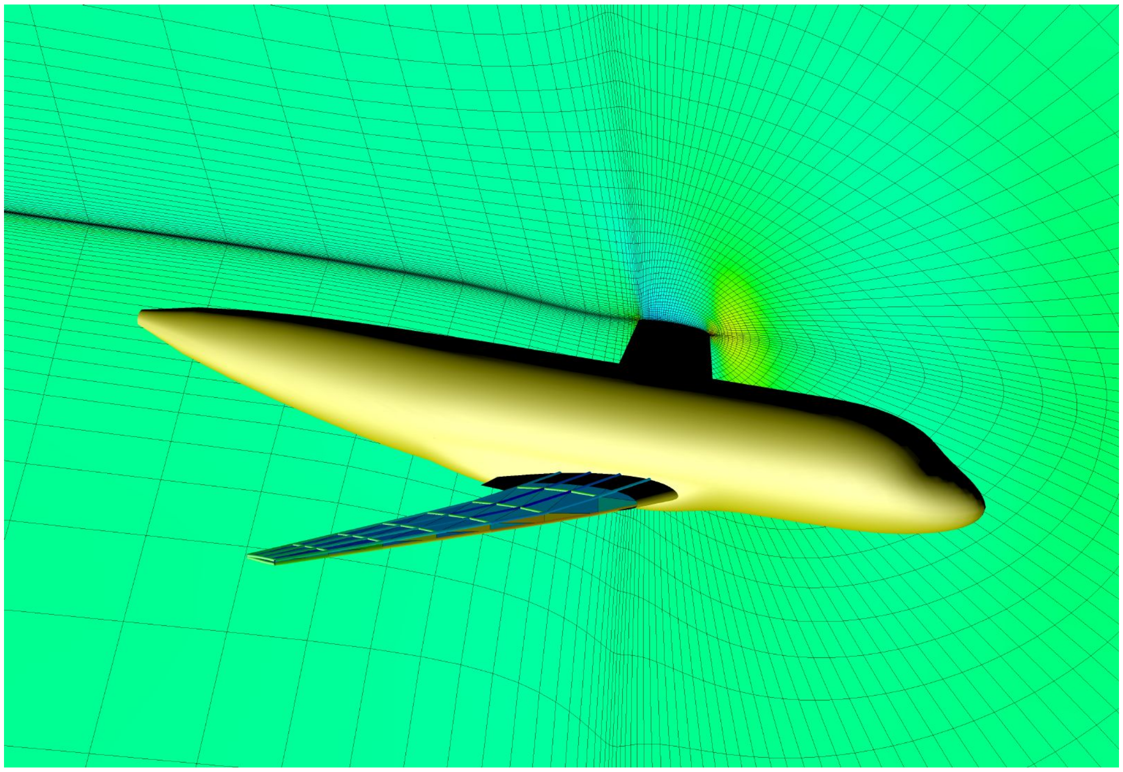

Figure 10: Aero-structural model and results.

[5] FPIG — Fortran to Python Interface Generator. http://cens.ioc.ee/projects/f2py2e/

[6] Numerical Extension to Python. http://numpy.sourceforge.net/

[7] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: a foundation for computer science.* Addison-Wesley, 1988

[8] P. Peterson. `f2py` - *Fortran to Python Interface Generator. Second Edition.* 2000 http://cens.ioc.ee/projects/f2py2e/usersguide.html

[9] Python Documentation: Extending and Embedding. http://www.python.org/doc/ext/

[10] P. Peterson. `PyFortranObject` *example usages.* 2001 http://cens.ioc.ee/projects/f2py2e/pyfobj.html

[11] Reuther, J., J. J. Alonso, J. R. R. A. Martins, and S. C. Smith. "A Coupled Aero-Structural Optimization Method for Complete Aircraft Configurations", *Proceedings of the 37th Aerospace Sciences Meeting*, AIAA Paper 1999-0187. Reno, NV, January, 1999