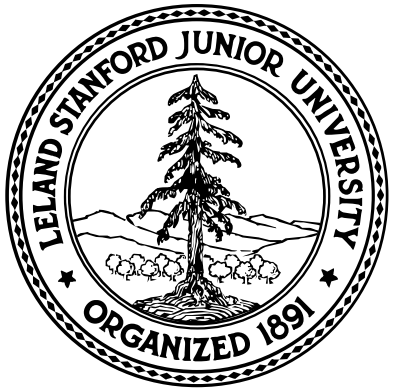
FORTRAN TO PYTHON INTERFACE GENERATOR WITH AN APPLICATION TO AEROSPACE ENGINEERING

Pearu Peterson

Institute of Cybernetics at TTU
Estonia

Joaquim R. R. A. Martins
Juan J. Alonso



Dept. of Aeronautics and Astronautics
Stanford University, CA

Outline

- Motivation for using Python to wrap Fortran
- What is f2py and how does it work?
- Simple example
- f2py features
- Future work
- Engineering application
- Conclusions

Motivation

- Fortran is still here:
  - Large number of numerical libraries and legacy code
  - Programming language of choice in most engineering applications
  - Very efficient for numerical applications
- Often necessary to "glue" different applications
  - Need for scripting, file parsing or object-oriented features
  - Not necessarily in the same language
- Python is a good choice that fulfills these needs.
- f2py automates the wrapping process.

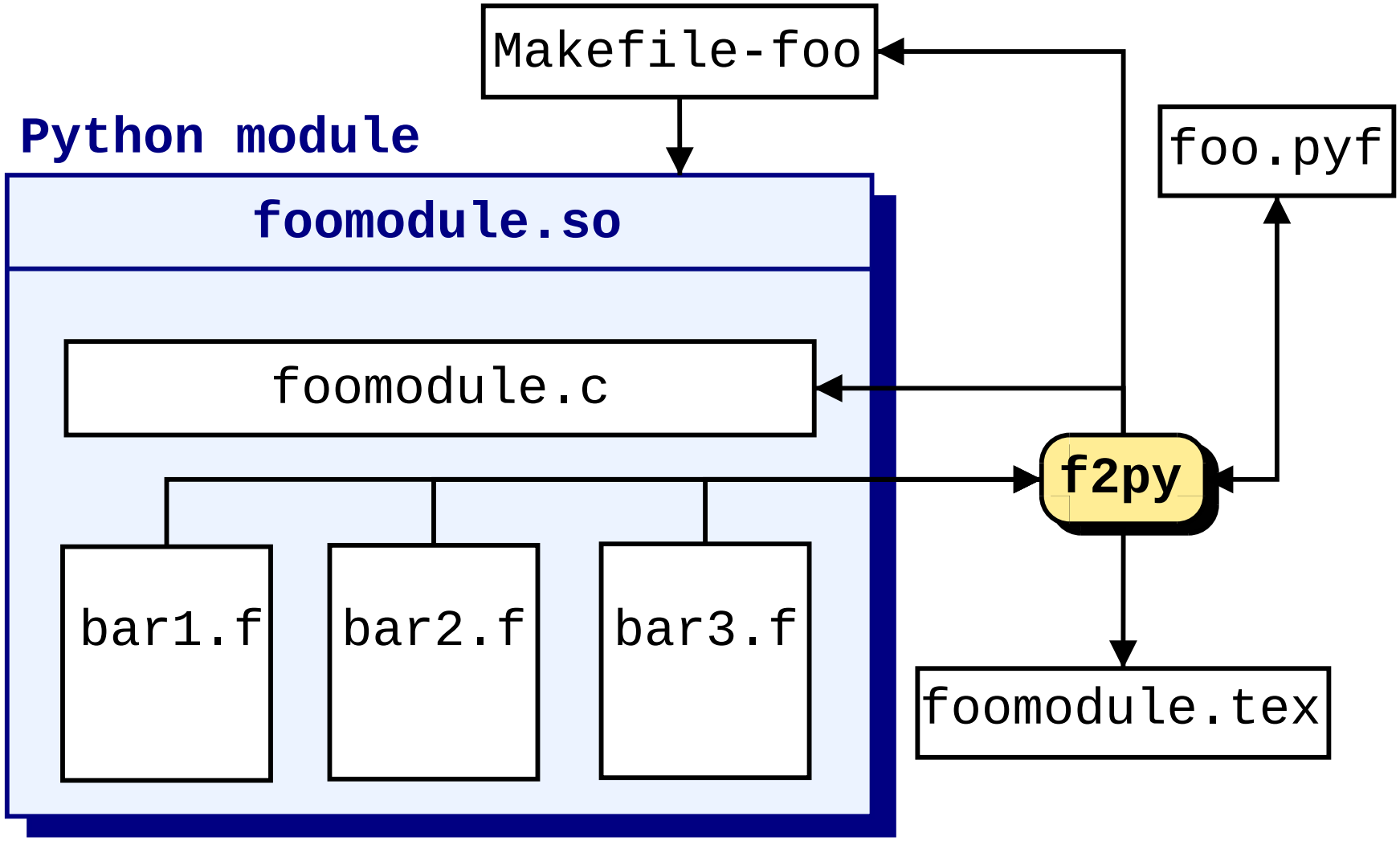
What is f2py?

Tool for creating Fortran to Python interfaces.

Wrappers created with f2py enable the following operations:

- Call Fortran routines from Python
  - Fortran 77/90/95 routines
  - Fortran 90/95 module routines
- Access Fortran data from Python
  - Fortran 77 common block variables
  - Fortran 90/95 module variables
- Call Python functions from Fortran

How Does it Work?



Example: Wrapping Simple Fortran 77 Routines

1. Generate interface
   ```
   > f2py foo.f bar.f -m
   foobar -h foobar.pyf
   ```

2. Edit signature file, `foobar.pyf` (optional).

3. Build module
   ```
   > make -f Makefile-foobar
   ```

4. Python session
   ```
   >>> import foobar
   ...
   ```

```
! Fortran file foo.f:
      subroutine foo(a)
      integer a
      a = a + 5
      end
```

```
! Fortran file bar.f:
      function bar(a,b)
      integer a,b,bar
      bar = a + b
      end
```

Step 1: Generate Interface

```
> f2py foo.f bar.f -m foobar -h foobar.pyf

Reading fortran codes...
        Reading file 'foo.f'
        Reading file 'bar.f'
Post-processing...
        Block: foobar
                        Block: foo
                        Block: bar
Saving signatures to file "foobar.pyf"
Creating 'Makefile-foobar'...
        Linker: ld ('Linker for MIPSpro 7 Compilers' 7.30.)
        Fortran compiler: f77 ('MIPSpro 7 Compilers' 7.30)
        C compiler: cc ('MIPSpro 7 Compilers' 7.30)
Stopping. Edit the signature file and then run f2py on the signature
        file: f2py foobar.pyf
        Or run GNU make to build shared module: gmake -f Makefile-<modulename>
```

Step 2 (Optional): Edit Signature File, `foobar.pyf`

```
!%f90 -*- f90 -*-
module foobar ! in
    interface  ! in :foobar
        subroutine foo(a) ! in :foobar:foo.f
            integer :: a
        end subroutine foo
        function bar(a,b) ! in :foobar:bar.f
            integer :: a
            integer :: b
            integer :: bar
        end function bar
    end interface
end module foobar

! This file was auto-generated with f2py (version:1.218).
! See http://cens.ioc.ee/projects/f2py2e/
```

Step 3: Build Module

```
> gmake -f Makefile-foobar

/usr/bin/f2py foobar.pyf
Reading fortran codes...
        Reading file 'foobar.pyf'
Post-processing...
        Block: foobar
                        Block: foo
                        Block: bar
Keeping existing 'Makefile-foobar'. Use --overwrite-makefile to overwrite.
Building modules...
        Building module "foobar"
                Constructing wrapper function "foo"
                  foo(a)
                Constructing wrapper function "bar"
                  bar = bar(a,b)
        Wrote C/API module "foobar" to file "foobarmodule.c"
        Documentation is saved to file "foobarmodule.tex"
Run GNU make to build shared modules: gmake -f Makefile-<modulename> [test]
cc -mips4 -n32  -I/usr/local/include/python1.5/Numeric -I/usr/local/include/python1.5
-c -o foobarmodule.o foobarmodule.c
f77 -mips4 -n32     -c -o foo.o foo.f
f77 -mips4 -n32     -c -o bar.o bar.f
ld -shared -s  -o foobarmodule.so foobarmodule.o  foo.o bar.o -L/usr/lib32 -lftn -lc -lfortran
```

Step 4: Python Session

```
>>> import foobar
>>> print foobar.__doc__
This module 'foobar' is auto-generated with f2py (version:1.218).
The following functions are available:
  foo(a)
  bar = bar(a,b)
.
>>> print foobar.foo.__doc__
Function signature:
  foo(a)
Required arguments:
  a : input int

>>> print foobar.bar(2,3)
5
>>> from Numeric import *
>>> a = array(3)
>>> print a,foobar.foo(a),a
3 None 8
```

Main Features

1. Support for all Fortran intrinsic types.

2. Support for different types of dimension specifications, e.g.:
```
real a(5), b(3:8), c(*)
integer n
parameter (n=3)
real d(n,5)
```

3. Call Fortran 77/90/95 routines and Fortran 90/95 module routines.

4. Access Fortran 77 common blocks.

5. Access Fortran 90/95 module data, including allocatable data.

Main Features

6. Supported compilers:
   - GNU project C Compiler (gcc)
   - Compaq Fortran
   - VAST/f90 Fortran
   - Absoft f77/f90
   - MIPSpro 7 Compilers
7. Supported platforms:
   - Intel/Alpha Linux
   - HP-UX
   - IRIX64
8. Documentation:
   - f2py User's Guide
   - Mailing list: `f2py-users@cens.ioc.ee`
   - Homepage: `http://cens.ioc.ee/projects/f2py2e`
   - Development under CVS, GNU LGPL.

Supported Argument Attributes

- `intent(<intent>)`: `in`, `out`, `inout`, `hide` or a combination.
- `dimension(<dimspec>)`
- `depend([<names>])`: dependency on arguments in `<names>`.
- `check([<C booleanexpr>])`: verify the size 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.

Accessing Fortran 77 Common Block Data

Python:

```
>>> import foo
>>> print foo.__doc__
Functions:
  bar()
COMMON blocks:
  /fun/ i,r(4)
.
>>> foo.fun.r=range(5)
>>> foo.fun.i=13
>>> foo.bar()
 i= 13
 r=  0.  1.  2.  3.
>>> print foo.fun.__doc__
i - 'i'-scalar
r - 'f'-array(4)
```

Fortran:

```
!Fortran file foo.f:
      subroutine bar()
      integer i
      real r(4)
      common /fun/ i,r
      write(*,*) "i=",i
      write(*,*) "r=",r
      end
```

Accessing Fortran 90 Module Data

Python:

```
>>> import foo
>>> print foo.__doc__
Functions:
  bar()
Fortran 90/95 modules:
  fun --- r,i.
>>> foo.fun.r = range(4)
>>> foo.bar()
 i= 0
 r= 0.E+0,  1.,  2.,  3.
>>> foo.fun.i = 17
>>> foo.fun.r = range(7)
>>> foo.bar()
 i= 17
 r= 0.E+0,  1.,  2.,  3.,  4.,  5.,  6.
```

Fortran:

```
!Fortran file fun.f90
module fun
  integer i
  real, allocatable :: r(:)
end module fun
subroutine bar()
  use fun
  write(*,*) "i=",i
  write(*,*) "r=",r
end subroutine bar
# re-allocation
```

Calling Python Functions from Fortran

Python:

```
>>> def bar(i,a):
...       print 'i=',i
...       a[:] = [1,3,2,5,7]
>>> foo.fun(bar)
i= 4
a= 1.0 3.0 2.0 5.0 7.0
>>> print foo.fun.__doc__
fun - Function signature:
  fun(bar,[bar_extra_args])
Required arguments:
  bar : call-back function
Optional arguments:
  bar_extra_args := () input tuple
Call-back functions:
  def bar(e_4_e,a): return
  Required arguments:
    e_4_e : input int
    a : input rank-1 array('f') with bounds (5)
```

Fortran:

```
subroutine fun(bar)
external bar
real a(5)
call bar(4,a)
write(*,*) "a=",a
end
```

Plans for the Future

1. Use `distutils`, replacing current Makefile approach. Need Fortran compiler support from `distutils`.

2. Support for MS Windows? Very little familiarity with Fortran compilers under Windows.

3. Find a better solution for C-Fortran array transpose problem.

4. GUI for manipulating signature files (a good excuse to learn wxPython).

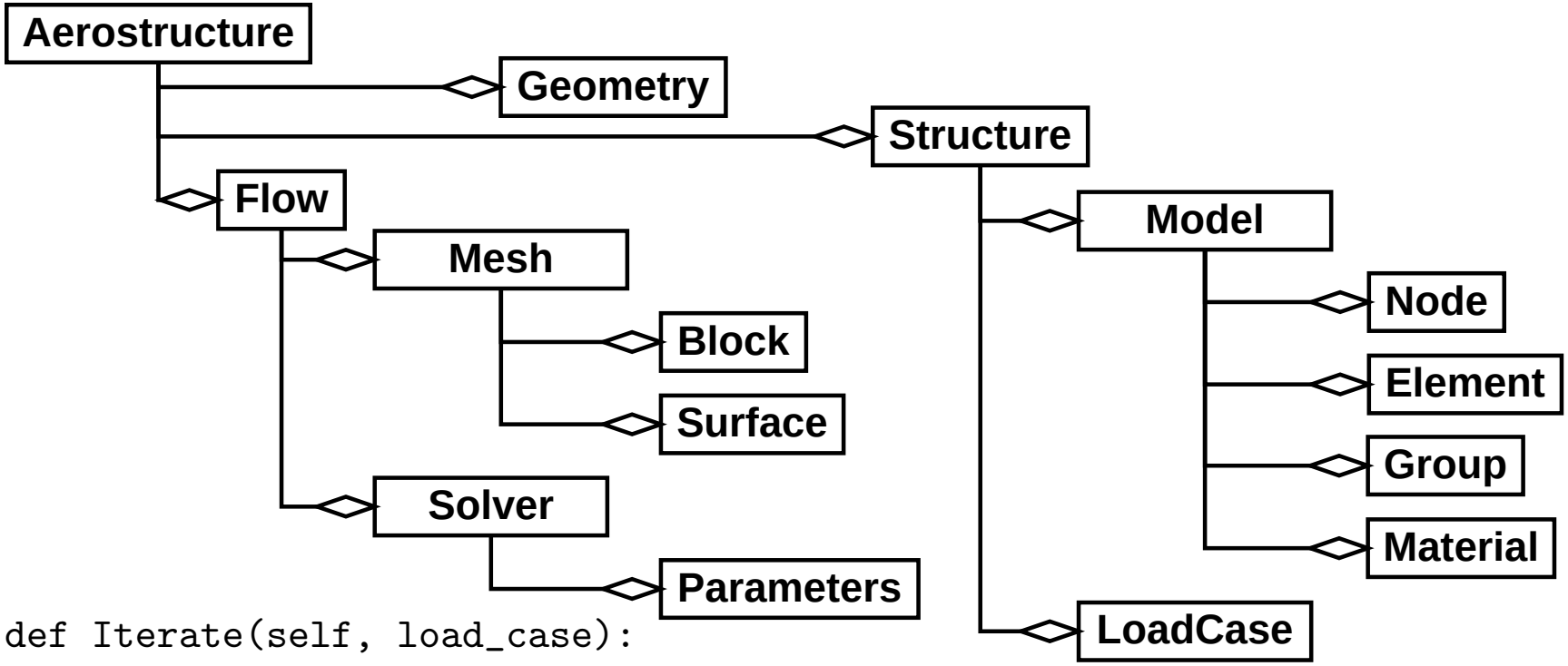
5. Maintain a repository for signature files.

6. f2py – 3rd Edition.

Application to Aerospace Engineering

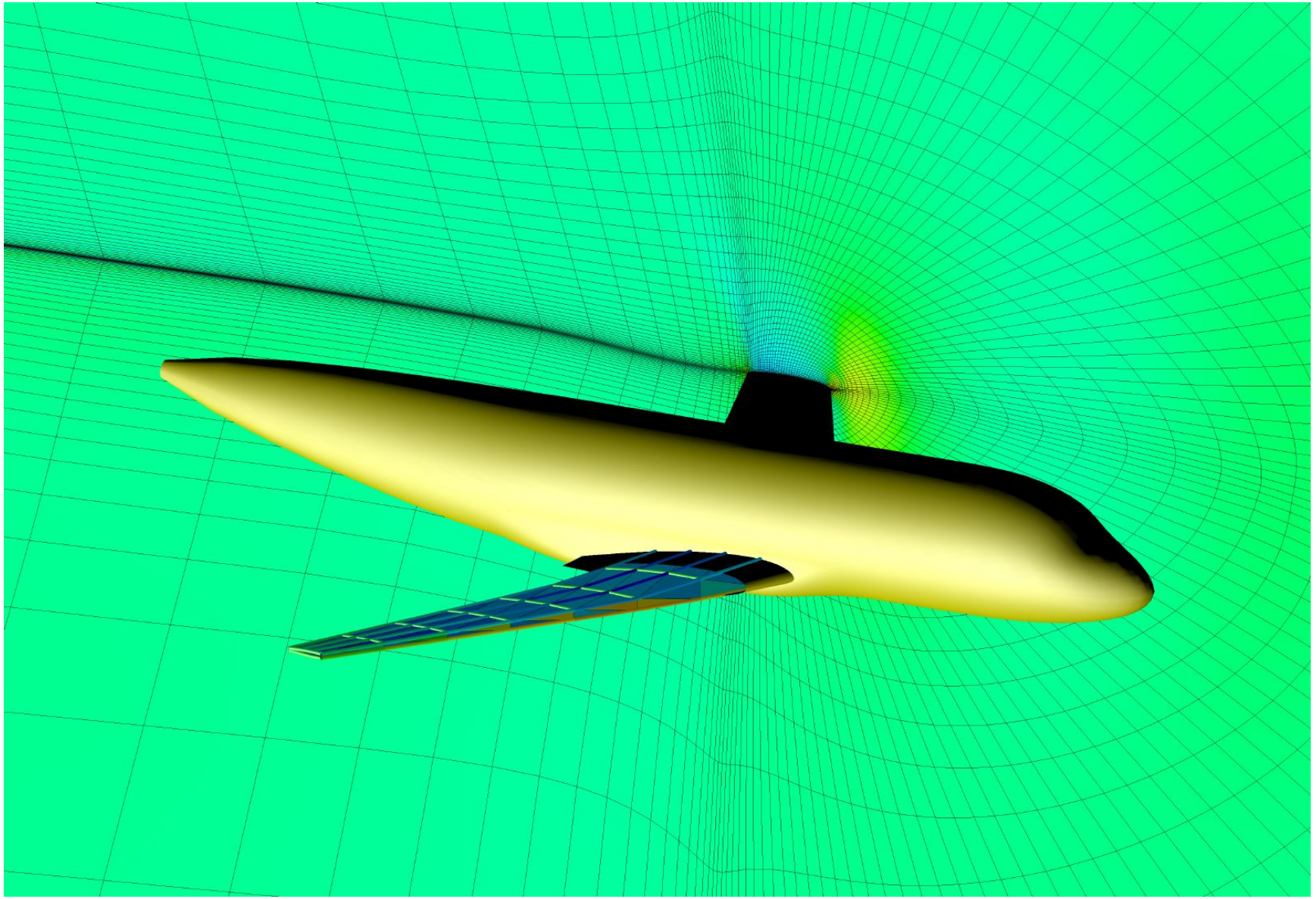
- Aero-structural design optimization framework:
  - 3D Computational fluid dynamics
  - Structural finite-element model
  - Geometry engine and database
- Motivation:
  - User friendly interface for using large Fortran solvers.
  - A better way for scripting or gluing Fortran programs.
  - Object-oriented framework to facilitate extensibility for more complex design problems.
- Requirements:
  - Access to Fortran 77 common blocks and Fortran 90 modules.
  - Wrapping process as automated as possible.

Python Module Design




```
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
```

Aero-Structural Model and Solution



Conclusions

- Presented a tool for creating Python wrappers for Fortran programs automatically.
- The tool was used in a complex aerospace engineering application.
- In practice, f2py was shown to provide:
  - An extremely versatile interface
  - A high level of automation
  - Robustness
- f2py has the potential to become the standard tool for wrapping Fortran with Python.