Introduction to F2Py

Goddard Python User's Group

pythonbootcamp@bigbang.gsfc.nasa.gov

Goddard Space Flight Center

June 17, 2016

Table of contents

Useful Links

- **Reference Document:** http://www.scipy.org/F2py
- **User's Guide and Reference Manual:** http://cens.ioc.ee/projects/f2py2e/usersguide/index.html
- **Frequently Asked Questions:** http://cens.ioc.ee/projects/f2py2e/FAQ.html

Basic Facts

- Python scripts are powerful and fast to write
- Python can be too slow to do intensive calculations
- Programs using low level languages such as Fortran and C are fast for computing but slow to write.
- Use the best of the two worlds: write most of the programs in Python and only write the calculations in a fast low level language.

What is F2Py?

- Fortran to Python interface generator
- Reuse available Fortran code within Python
- Extend Python with high-performance computational modules
- Suitable for wrapping C libraries to Python

F2Py Features

1. Scans Fortran codes for subroutine/function/data signatures
2. Call Fortran 77/90/95 modules and C functions from Python
3. Access Fortran 77 COMMON blocks and Fortran 90 module data (also allocatable arrays) from Python
4. Call Python functions from Fortran and C (callbacks)
5. Handle Fortran/C data storage issues
6. Generate documentation strings
7. Is part of Numpy

Limitations

- Meets the Fortran 95 programming standards
- Does not support:
  1. Derived types
  2. Pointers

Work is under way to make such support available (with G3 F2Py) and to meet the Fortran 2003 standards.

Main F2Py Command Line Options

```
--fcompiler=      Specify Fortran compiler type by vendor
--compiler=       Specify C compiler type

--help-fcompiler  List available Fortran compilers and exit
--f77exec=        Specify the path to F77 compiler
--f90exec=        Specify the path to F90 compiler
--f77flags=       Specify F77 compiler flags
--f90flags=       Specify F90 compiler flags
--opt=            Specify optimization flags
--debug           Compile with debugging information
```

Some Supported Compilers

```
Key         Description of compiler
-------------------------------
g95         G95 Fortran Compiler
gnu         GNU Fortran 77 compiler
nag         NAGWare Fortran 95 Compiler
pg          Portland Group Fortran Compiler
absoft      Absoft Corp Fortran Compiler
compaq      Compaq Fortran Compiler
intel       Intel Fortran Compiler for 32-bit apps
intele      Intel Fortran Compiler for Itanium apps
intelem     Intel Fortran Compiler for EM64T-based apps
lahey       Lahey/Fujitsu Fortran 95 Compiler
hpux        HP Fortran 90 Compiler
ibm         IBM XL Fortran Compiler
intelev     Intel Visual Fortran Compiler for Itanium apps
```

What F2Py Does

- F2Py takes a Fortran subroutine and some additional intructions
- F2Py compiles the Fortran source code and builds a module (dynamic library which contains native machine code)
- The module is imported into a Python code and utilized there as a regular Python module.

Initial Preparation

1. In all the subroutines you want to pass to Python, remove anything related to pointers and derived types
2. Change the main program into a subroutine

Sample Test Case

```
1        subroutine matrixMult(C, A, B, n)
2
3        implicit none
4
5        integer, intent(in)  :: n
6        real*8,  intent(in)  :: A(n,n)
7        real*8,  intent(in)  :: B(n,n)
8        real*8,  intent(out) :: C(n,n)
9
10       C = matmul(A,B)
11
12       return
13
14       end subroutine matrixMult
```

Method 1: Using F2Py within the Python Code

- Use F2Py available in Numpy
- Everything is done within the Python code where you want to use the module generated by F2Py
  1. Open the Fortran source file
  2. Compile the Fortran source file with F2Py
  3. Import the generated module

Simple Test Case

```
#!/usr/bin/env python
import numpy as np
import numpy.f2py as f2py
...
fid = open('forMatMul_ref.f90')
source = fid.read()
fid.close()
f2py.compile(source, modulename='forMatMul')
import forMatMul
...
AB = forMatMul.matrixmult(A,B)
```

Method 2: Changing the Fortran Source Code

- This is more important in Fortran 77 that does not have the `INTENT` declaration.
- Consider all the arguments of the subroutine you want to call within Python.
- Add command strings for F2Py having the form !f2py to specify the intent of each argument.

The Modified Test Case

```
1         subroutine matrixMult(C, A, B, n)
2         implicit none
3         real*8 A(n,n)
4         real*8 B(n,n)
5         real*8 C(n,n)
6         integer n
7  !f2py  intent(out) :: C
8  !f2py  intent(in)  :: A
9  !f2py  intent(in)  :: B
10 !f2py  intent(in)  :: n
11
12        C = matmul(A,B)
13
14        return
15        end subroutine matrixMult
```

Important Intent Specifications

```
intent(in)                  input variable
intent(out)                 output variable
intent(in,out)              input and output variable
intent(in,hide)             hide from argument list
intent(in,hide,cache)       keep hidden allocated arrays in memory
intent(in,out,overwrite)    enable an array to be overwritten (if feas
intent(in,ou,copy)          disable an array to be overwritten
depend(m,n) q               make q's dimensions depend on m and n
```

Run F2Py

```
f2py -m moduleName -c --fcompiler=g95 \
     file1.f90 file2.f90 only: routine1 routine2 routine3
```

Method 3: Generate a Signature File

F2Py can create a signature file that determines the interfaces of the functions/subroutines in the module to be created. You need to issue the command:

```
f2py -m moduleName -h signatureFile.pyf listOfFortranFiles
```

You can edit the signature file (*signatureFile.pyf*) to:

- Comment out any subroutine having in its argument list a variable declared as *dimension(:)*.
- Add intentions that are not legal in Fortran. We adjust the text `intent(in)` to `intent(in,hide)`.

Sample Signature File

```
1  python module forMatMul
2   interface
3    subroutine matrixmult(c,a,b,n)
4     real*8 dimension(n,n),intent(out),depend(n,n) :: c
5     real*8 dimension(n,n),intent(in) :: a
6     real*8 dimension(n,n),intent(in),depend(n,n) :: b
7     integer optional,intent(in), &
8         check(shape(a,0)==n),depend(a) :: n=shape(a,0)
9    end subroutine matrixmult
10  end interface
11 end python module forMatMul
```

Edited Signature File

```
1  python module forMatMul
2   interface
3    subroutine matrixmult(c,a,b,n)
4     real*8 dimension(n,n),intent(out),depend(n,n) :: c
5     real*8 dimension(n,n),intent(in) :: a
6     real*8 dimension(n,n),intent(in),depend(n,n) :: b
7     integer optional,intent(in,hide), &
8         check(shape(a,0)==n),depend(a) :: n=shape(a,0)
9    end subroutine matrixmult
10  end interface
11 end python module forMatMul
```

With the `hide` statement, the integer `n` no longer has to be passed in the argument list.

Generate the Module

Issue the command:

```
f2py -c --fcompiler=gnu95 signatureFile.pyf listOfFortranFiles
```

Using the Module in a Python Script

```
#!/usr/bin/env python
...
import sys
...
sys.path.append('...')
import forMatMul
...
...
AB = forMatMul.matrixmult(A, B)
```

Useful Compilation Options

Printing Detailed Information

```
f2py -c --debug-capi --fcompiler=gnu95 signatureFile.pyf \
      listOfFortranFiles
```

Linking with External Libraries

```
f2py -m moduleName -h signatureFile.pyf listOfFortranFiles \
      only: routine1 routine2 routine3

f2py -c --fcompiler=gnu95 signatureFile.pyf \
      listOfFortranFiles \
      -L/PathToLibrary -lLibName
```

Python Script for Matrix Multiplication

```
1  #!/usr/bin/env python
2  import numpy as np
3  from time import *
4  import sys
5  import forMatMul
6
7  n = n = int(sys.argv[1])
8
9  A = np.random.rand(n,n)
10 B = np.random.rand(n,n)
11
12 begTime = time()
13 AB = forMatMul.matrixmult(A,B)
14 endTime = time()
```

Performance of Matrix Multiplication

	$n = 4000$	$n = 5000$	$n = 6000$
Numpy (built with MKL 15)	5.63	11.43	18.82
F2Py (using matmult)	73.88	149.51	265.32
Fortran (using matmult)	74.04	147.90	278.83
Fortran (using MKL 15)	5.27	10.26	18.21

Fortran Subroutine for Jacobi Iteration

```
1         subroutine timeStep(u,n,error)
2         double precision u(n,n), error
3         integer n,i,j
4   !f2py intent(in,out) :: u
5   !f2py intent(out) :: error
6   !f2py intent(in) :: n
7         double precision tmp, diff
8         error = 0d0
9         do j=2,n-1
10          do i=2,n-1
11            tmp = u(i,j)
12            u(i,j)=(4.0d0*(u(i-1,j)+u(i+1,j)+u(i,j-1) &
13                   + u(i,j+1))+u(i-1,j-1) + u(i+1,j+1) &
14                   + u(i+1,j-1)+ u(i-1,j+1))/20.0d0
15            diff = u(i,j) - tmp
16            error = error + diff*diff
17          end do
18        end do
```

Python Script for the Jacobi Iteration

```
1  import timeStep
2  j=numpy.complex(0,1); nPoints=100
3  u=numpy.zeros((nPoints,nPoints),dtype=float)
4  pi_c=float(math.pi)
5  x=numpy.r_[0.0:pi_c:nPoints*j]
6  u[0,:]=numpy.sin(x); u[nPoints-1,:]=numpy.sin(x)
7
8  def solve_laplace(u,nPoints):
9      iter =0
10     err = 2
11     while(iter <1000000 and err>1e-6):
12         (u,err)=timeStep.timestep(u,nPoints)
13         iter+=1
14     return (u,err,iter)
15
16 (u, err, iter) = solve_laplace(u,nPoints)
```

Performance of the Jacobi Iteration

	$n = 350$	$n = 400$	$n = 450$
Numpy (MKL 15)	483.36	809.82	1285.706
F2Py	166.18	280.89	446.98
Fortran	74.76	121.19	182.86

Description of the Application

We have a Fortran 90 code that attempts to numerically solve the two dimensional convection-diffusion equation with constant coefficients:

$$\begin{aligned} u_{xx} + u_{yy} + \sigma u_x + \tau u_y &= f(x,y), \qquad (x,y) \in \Omega \\ u(x,y) &= g(x,y), \qquad (x,y) \in \partial\Omega \end{aligned}$$

where Ω is a convex domain and $\partial\Omega$ is the boundary of Ω.

The equation is discretized using a fourth-order compact finite difference scheme (9-point stencil) and the multigrid method is employed as iterative solver.

Fortran Source Code

The entire code contains:

- 9 files (including the main program)
- 14 subroutines
- 1 module
- 2 2D global variables and 1 1D global variable

Main Driver of the Fortran Code

```
1  SUBROUTINE MGSP2D( NX, NY, H, TOL, IO, Q, LQ, RN, &
2                     IERR, SIG, TAU )
3
4  USE MG_levelsMod
5
6  integer, intent(in)     :: NX, NY, LQ
7  REAL*8,  intent(in)     :: SIG, TAU
8  REAL*8,  intent(in)     :: H, TOL
9  integer, intent(in)     :: IO
10 integer, intent(out)    :: IERR
11 REAL*8,  intent(out)    :: RN
12 REAL*8 , intent(inOut)  :: Q(LQ)
```

What We Want To Achieve

1. Write the Fortran main program as a subroutine
2. Use F2Py to generate a module that will be used in Python
3. Write a Python script that:
   - Does all the initializations
   - Calls the main driver of the multigrid method (available in the module created by F2Py)
   - Computes the maximum error of the approximated solution.
   - Plots the solution using Matplotlib

Shell Script

```
#!/bin/csh -f

# Make sure that Python is loaded

f2py --debug-capi -m MGconvDiff2d -h sgnFile.pyf MG*.F90
f2py -c --fcompiler=gnu95 --debug-capi sgnFile.pyf MG*.F90

./f2py_MGconvDiff2d.py 16
```

Generated Signature File

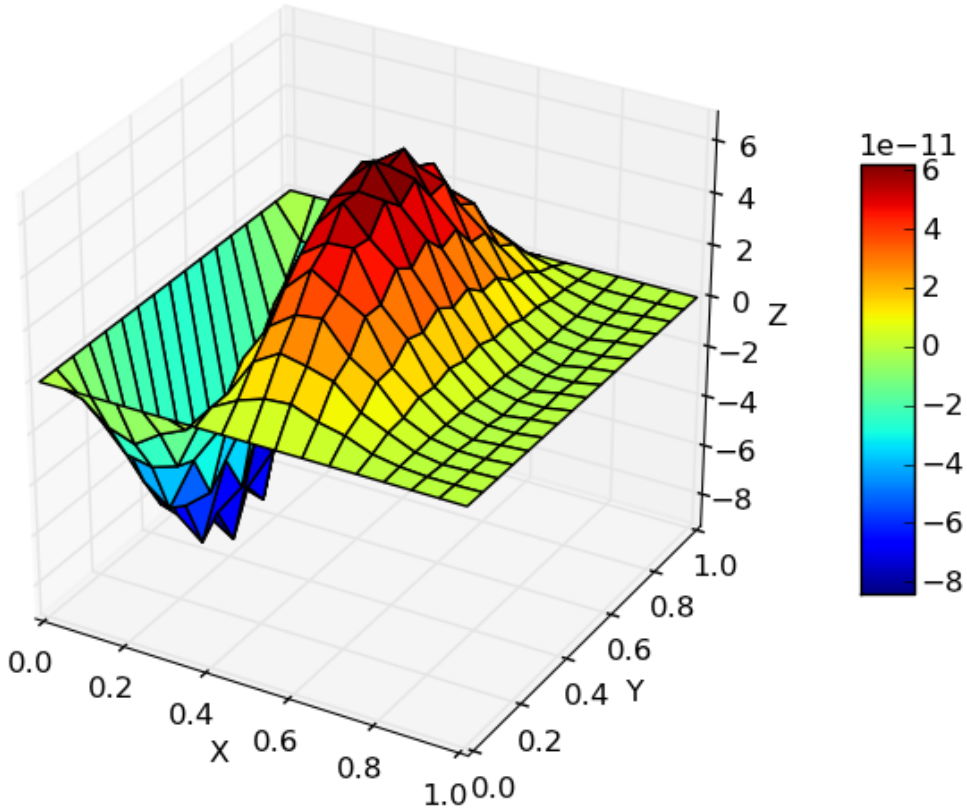
```
1  subroutine mgsp2d(nx,ny,h,tol,io,q,lq,rn,ierr,sig,tau)
2    use mg_levelsmod
3    integer intent(in) :: nx
4    integer intent(in) :: ny
5    real*8 intent(in) :: h
6    real*8 intent(in) :: tol
7    integer intent(in) :: io
8    real*8 dimension(lq),intent(inOut) :: q
9    integer optional,intent(in),check(len(q)>=lq), &
10                          depend(q) :: lq=len(q)
11   real*8 intent(out) :: rn
12   integer intent(out) :: ierr
13   real*8 intent(in) :: sig
14   real*8 intent(in) :: tau
15 end subroutine mgsp2d
```

Overview of the Python Code

```
1  import MGconvDiff2d
2  ...
3  Q = np.zeros((LQ),dtype=float)
4
5  Q[0:MSIZE]         = U.reshape(MSIZE)
6  Q[MSIZE:2*MSIZE] = F.reshape(MSIZE)
7
8  # Set the multigrid grid structure
9  MGconvDiff2d.mg_levelsmod.setgridstructure(NX, NY, \
10                                            LQ, A, B)
11
12 # Call the multigrid solver
13 RN, IERR = MGconvDiff2d.mgsp2d(NX, NY, H, TOL, IO, \
14                                 Q, SIG, TAU)
15
16 U = Q[0:MSIZE].reshape(NX+1,NY+1)
```

Plot of the Solution

Approximated solution on a 16×16 grid when the exact solution is $u(x, y) = 0.0$.



Things to Consider

- F2Py is great when dealing with one subroutine only. When many subroutines are involved careful consideration is required.
- Avoid using `EQUIVALENCE` statements.
- If `COMMON BLOCKS` are shared among subroutines, it might be easier to make them available through include files.
- As far as possible, simplify the argument list of the routines that will be call within Python.
- Understanding the signature file syntax is important to simplify the wrapper and fix potential problems.
- Fortran 77 subroutines lack the argument intent information. Editing the signature file may be required to add the intent statements.

References I


J.R. Johansson, *Using Fortran and C code with Python*, http://nbviewer.ipython.org/urls/raw.github.com/-jrjohansson/scientific-python-lectures/master/-Lecture-6A-Fortran-and-C.ipynb, 2013.


Pearu Peterson, *F2PY: a tool for connecting Fortran and Python programs*, *Int. J. Computational Science and Engineering*, Vol. 4, No. 4, p. 296–305 (2009).


Pierre Schnizer, *A Short Introduction fo F2PY*, http://dsnra.jpl.nasa.gov/software/Python/-F2PY_tutorial.pdf, 2002.


J. Kouatchou, *F2Py and Static Libraries*, https://modelingguru.nasa.gov/docs/DOC-2343.

References II