An Introduction to F2Py

Jules Kouatchou, Hamid Oloso and Mike Rilee

Jules.Kouatchou@nasa.gov, Amidu.O.Oloso@nasa.gov and Michael.Rilee@nasa.gov

Goddard Space Flight Center
Software System Support Office
Code 610.3

April 29, 2013
Table of contents

1 Introduction

2 Methods for Creating Python Modules
   • Method 1
   • Method 2
   • Method 3

3 Two Simple Applications
   • Matrix Multiplication
   • Numerical Solution of the Laplace Equation

4 Real Application

5 Lessons Learned
Obtaining the Material

Slides for this session of the training are available from:

https://modelingguru.nasa.gov/docs/DOC-2322

You can obtain materials presented here on discover at

/discover/nobackup/jkouatch/pythonTrainingGSFC.tar.gz

After you untar the above file, you will obtain the directory pythonTrainingGSFC/ that contains:

Examples/
Slides/
We installed a Python distribution. To use it, you need to load the modules:

```
module load other/comp/gcc-4.5-sp1
module load lib/mkl-10.1.2.024
module load other/SIVO-PyD/spd_1.9.0_gcc-4.5-sp1
```
Useful Links

- **Reference Document**: http://www.scipy.org/F2py
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Description of compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>g95</td>
<td>G95 Fortran Compiler</td>
</tr>
<tr>
<td>gnu</td>
<td>GNU Fortran 77 compiler</td>
</tr>
<tr>
<td>nag</td>
<td>NAGWare Fortran 95 Compiler</td>
</tr>
<tr>
<td>pg</td>
<td>Portland Group Fortran Compiler</td>
</tr>
<tr>
<td>absoft</td>
<td>Absoft Corp Fortran Compiler</td>
</tr>
<tr>
<td>compaq</td>
<td>Compaq Fortran Compiler</td>
</tr>
<tr>
<td>intel</td>
<td>Intel Fortran Compiler for 32-bit apps</td>
</tr>
<tr>
<td>intele</td>
<td>Intel Fortran Compiler for Itanium apps</td>
</tr>
<tr>
<td>intelem</td>
<td>Intel Fortran Compiler for EM64T-based apps</td>
</tr>
<tr>
<td>lahey</td>
<td>Lahey/Fujitsu Fortran 95 Compiler</td>
</tr>
<tr>
<td>hpux</td>
<td>HP Fortran 90 Compiler</td>
</tr>
<tr>
<td>ibm</td>
<td>IBM XL Fortran Compiler</td>
</tr>
<tr>
<td>intelev</td>
<td>Intel Visual Fortran Compiler for Itanium apps</td>
</tr>
</tbody>
</table>
What F2Py Does

- F2Py takes a Fortran subroutine and some additional instructions
- 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

```fortran
subroutine matrixMult(C, A, B, n)

implicit none

integer, intent(in) :: n
real*8, intent(in) :: A(n,n)
real*8, intent(in) :: B(n,n)
real*8, intent(out) :: C(n,n)

C = matmul(A,B)

return

derived subroutine matrixMult
```
Method 1: Using F2Py within Python Code
How to Do It?

- 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

```python
#!/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: Change Source Code
Changes in 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 withing Python.
- Add command strings for F2Py having the form !f2py to specify the intent of each argument.
The Modified Test Case

```fortran
subroutine matrixMult(C, A, B, n)
implicit none
real*8  A(n,n)
real*8  B(n,n)
real*8  C(n,n)
integer n
! f2py intent(out) :: C
! f2py intent(in) :: A
! f2py intent(in) :: B
! f2py intent(in) :: n

C = matmul(A,B)

return
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 feasible)
- `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: Signature File
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:

\[ \text{f2py} \ -m \ \text{moduleName} \ -h \ \text{signatureFile.pyf} \ \text{listOfFortranFiles} \]

You can edit the signature file (\textit{signatureFile.pyf}) to:

- Comment out any subroutine having in its argument list a variable declared as \textit{dimension(:)}.
- Add intentions that are not legal in Fortran. We adjust the text \textit{intent(in)} to \textit{intent(in,hide)}.
Sample Signature File

```python
python module forMatMul

  interface

  subroutine matrixmult(c,a,b,n)
    real*8 dimension(n,n), intent(out), depend(n,n) :: c
    real*8 dimension(n,n), intent(in) :: a
    real*8 dimension(n,n), intent(in), depend(n,n) :: b
    integer optional, intent(in), &
    check(shape(a,0)==n), depend(a) :: n=shape(a,0)
  end subroutine matrixmult

end interface

end python module forMatMul
```
Edited Signature File

```python
python module forMatMul

  interface
    subroutine matrixmult(c,a,b,n)
      real*8 dimension(n,n), intent(out), depend(n,n) :: c
      real*8 dimension(n,n), intent(in) :: a
      real*8 dimension(n,n), intent(in), depend(n,n) :: b
      integer optional, intent(in, hide), &
      check(shape(a,0)==n), depend(a) :: n=shape(a,0)
    end subroutine matrixmult
  end interface

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

```python
#!/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 Matrix Multiplication

```python
#!/usr/bin/env python
import numpy as np
from time import *
import sys
import forMatMul

n = int(sys.argv[1])

A = np.random.rand(n,n)
B = np.random.rand(n,n)

begTime = time()
AB = forMatMul.matrixmult(A,B)
endTime = time()
```
## Performance of Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>$n = 1000$</th>
<th>$n = 1500$</th>
<th>$n = 2000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy (built with MKL 10)</td>
<td>8.19</td>
<td>28.5</td>
<td>75.2</td>
</tr>
<tr>
<td>Numpy (built with MKL 13)</td>
<td>0.25</td>
<td>1.21</td>
<td>1.38</td>
</tr>
<tr>
<td>F2Py (using matmult)</td>
<td>1.02</td>
<td>3.86</td>
<td>9.00</td>
</tr>
<tr>
<td>Fortran (using matmult)</td>
<td>1.07</td>
<td>3.67</td>
<td>8.81</td>
</tr>
<tr>
<td>Fortran (using MKL 13)</td>
<td>0.19</td>
<td>0.59</td>
<td>1.37</td>
</tr>
</tbody>
</table>
Fortran Subroutine for Jacobi Iteration

```fortran
subroutine timeStep(u,n,error)
  double precision u(n,n), error
  integer n,i,j
  ! f2py intent(in , out) :: u
  ! f2py intent( out) :: error
  ! f2py intent(in) :: n
  double precision tmp, diff
  error = 0d0
  do j=2,n-1
    do i=2,n-1
      tmp = u(i,j)
      u(i,j) = (4.0d0*(u(i-1,j)+u(i+1,j)+u(i,j-1) &
                  + u(i,j+1)) + u(i-1,j-1) + u(i+1,j+1) &
                  + u(i+1,j-1)+ u(i-1,j+1))/20.0d0
      diff = u(i,j) - tmp
      error = error + diff*diff
    end do
  end do
  error = sqrt(error)
end subroutine timeStep
```

Kouatchou, Oloso and Rilee
Python Script for the Jacobi Iteration

```python
import timeStep
j=numpy.complex(0,1); nPoints=100
u=numpy.zeros((nPoints,nPoints), dtype=float)
pi_c=float(math.pi)
x=numpy.r_[0.0:pi_c:nPoints*j]
u[0,:]=numpy.sin(x); u[nPoints-1,:]=numpy.sin(x)

def solve_laplace(u,nPoints):
    iter =0
    err = 2
    while (iter <1000000 and err>1e-6):
        (u,err)=timeStep.timestep(u,nPoints)
        iter+=1
    return (u,err,iter)

(u, err, iter) = solve_laplace(u,nPoints)
```
## Performance of the Jacobi Iteration

<table>
<thead>
<tr>
<th></th>
<th>$n = 100$</th>
<th>$n = 200$</th>
<th>$n = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numpy (MKL 10)</td>
<td>4.09</td>
<td>64.8</td>
<td>253.2</td>
</tr>
<tr>
<td>Numpy (MKL 13)</td>
<td>4.11</td>
<td>65.6</td>
<td>257.8</td>
</tr>
<tr>
<td>F2Py (no opt)</td>
<td>1.39</td>
<td>21.5</td>
<td>105.9</td>
</tr>
<tr>
<td>F2Py (with opt)</td>
<td>1.30</td>
<td>6.55</td>
<td>14.86</td>
</tr>
<tr>
<td>Fortran</td>
<td>1.37</td>
<td>5.67</td>
<td>12.85</td>
</tr>
</tbody>
</table>
We have a Fortran 90 code that attempts to numerically solve the two-dimensional convection-diffusion equation with constant coefficients:

\[ u_{xx} + u_{yy} + \sigma u_x + \tau u_y = f(x, y), \quad (x, y) \in \Omega \]

\[ u(x, y) = g(x, y), \quad (x, y) \in \partial\Omega \]

where \( \Omega \) is a convex domain and \( \partial\Omega \) is the boundary of \( \Omega \).

The equation is discretized using a fourth-order compact finite difference scheme (9-point stencil) and the multigrid method is employed as iterative solver.
The entire code contains:
- 9 files (including the main program)
- 14 subroutines
- 1 module
- 2 2D global variables and 1 1D global variable
SUBROUTINE MGSP2D( NX, NY, H, TOL, IO, Q, LQ, RN, & IERR, SIG, TAU )

USE MG_levelsMod

integer, intent(in) :: NX, NY, LQ
REAL*8, intent(in) :: SIG, TAU
REAL*8, intent(in) :: H, TOL
integer, intent(in) :: IO
integer, intent(out) :: IERR
REAL*8, intent(out) :: RN
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

source /usr/share/modules/init/csh

module purge
module load other/comp/gcc-4.5-sp1
module load lib/mkl-10.1.2.024
module load other/SIVO-PyD/spd_1.9.0_gcc-4.5-sp1

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

```fortran
subroutine mgsp2d(nx,ny,h,tol,io,q,lq,rn,ierr,sig,tau)
  use mg_levelsmod
  integer intent(in) :: nx
  integer intent(in) :: ny
  real*8 intent(in) :: h
  real*8 intent(in) :: tol
  integer intent(in) :: io
  real*8 dimension(lq),intent(inOut) :: q
  integer optional,intent(in),check(len(q)>=lq), &
    depend(q) :: lq=len(q)
  real*8 intent(out) :: rn
  integer intent(out) :: ierr
  real*8 intent(in) :: sig
  real*8 intent(in) :: tau
end subroutine mgsp2d
```
Overview of the Python Code

```python
import MGconvDiff2d

Q = np.zeros((LQ), dtype=float)

Q[0:MSIZE] = U.reshape(MSIZE)
Q[MSIZE:2*MSIZE] = F.reshape(MSIZE)

# Set the multigrid grid structure
MGconvDiff2d.mg_levelsmod.setgridstructure(NX, NY, 
                                          LQ, A, B)

# Call the multigrid solver
RN, IERR = MGconvDiff2d.mgsp2d(NX, NY, H, TOL, IO, 
                                Q, SIG, TAU)

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` statement
- 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 II
