

Wrapping Fortran Code for Python with F2PY

Neilen Marais

CEMAGG Group
Electrical and Electronic Engineering Department
University of Stellenbosch
South Africa

`nmarais@gmail.com`

CTPUG, September 2006

Outline

- ① Using Python with External Compiled Code
- ② Using F2PY to Wrap Fortran Code
- ③ Retroactive Application of Object Oriented Design

Introduction

Two main uses of Fortran wrappers:

Wrapping Numerical Libraries

Extending Existing Applications

Introduction

Two main uses of Fortran wrappers:

Wrapping Numerical Libraries

- SciPy does this extensively
- Quite simple
- Just pass arrays in, get them out

Extending Existing Applications

Introduction

Two main uses of Fortran wrappers:

Wrapping Numerical Libraries

- SciPy does this extensively
- Quite simple
- Just pass arrays in, get them out

Extending Existing Applications

- Benefit from Python/SciPy features
- Sharing of data-structures
- Put OO face on Fortran data
- Fortran calling Python routines

Outline

- ① Using Python with External Compiled Code
- ② Using F2PY to Wrap Fortran Code
- ③ Retroactive Application of Object Oriented Design

Why Extend Fortran Apps with Python Wrappers

- Not straight forward to apply object oriented design
 - ▶ Not coded in an OO language
- Legacy code is very valuable!
 - ▶ Long term investment
 - ▶ Trusted, tested and known to work

Why Extend Fortran Apps with Python Wrappers

- Not straight forward to apply object oriented design
 - ▶ Not coded in an OO language
- Legacy code is very valuable!
 - ▶ Long term investment
 - ▶ Trusted, tested and known to work

Three Options for the Future

- 1 Continue as before
 - ▶ Low short term risk
 - ▶ Potential long term cost

Why Extend Fortran Apps with Python Wrappers

- Not straight forward to apply object oriented design
 - ▶ Not coded in an OO language
- Legacy code is very valuable!
 - ▶ Long term investment
 - ▶ Trusted, tested and known to work

Three Options for the Future

- ① Continue as before
 - ▶ Low short term risk
 - ▶ Potential long term cost
- ② From scratch re-write
 - ▶ Risky and costly!

Why Extend Fortran Apps with Python Wrappers

- Not straight forward to apply object oriented design
 - ▶ Not coded in an OO language
- Legacy code is very valuable!
 - ▶ Long term investment
 - ▶ Trusted, tested and known to work

Three Options for the Future

- ① Continue as before
 - ▶ Low short term risk
 - ▶ Potential long term cost
- ② From scratch re-write
 - ▶ Risky and costly!
- ③ Wrap existing code using a VHLL
 - ▶ Lower risk
 - ▶ Added advantages

Using External Compiled Code with Python

Python C-API

- Been around since just about the start of time
- User has to integrate with Python's memory management
- Quite a lot of work
- Error prone

Using External Compiled Code with Python

Python C-API

- Been around since just about the start of time
- User has to integrate with Python's memory management
- Quite a lot of work
- Error prone

Automatic Wrapper Generators

- Removes most of the manual labour
- F2PY and PyFort generate Fortran wrappers for Python
- F2PY now part of SciPy
- F2PY seems more actively maintained

Outline

- ① Using Python with External Compiled Code
- ② Using F2PY to Wrap Fortran Code
- ③ Retroactive Application of Object Oriented Design

Wrapping Fortran using F2PY

- Automatically generate Python interfaces to Fortran code
- Full F77 support
- Partial F90 support
 - ▶ Module data
 - ▶ Module subroutines
 - ▶ Allocated memory
- Simple wrappers in F90 code compensate

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
  CONTAINS

  SUBROUTINE process_data(factors, n)
    USE data
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i
    DO i = 1,n
      test_arr = test_arr - (-1)**i &
        * test_arr**factors(i)
    END DO
  END SUBROUTINE process_data

END MODULE prog
```

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data  
  IMPLICIT NONE
```

```
END MODULE data
```

test_data.f90

- MODULE data

test_prog.f90

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

- MODULE prog

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
CONTAINS
```

```
END MODULE prog
```

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

- MODULE prog
- SUBROUTINE process_data

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
  CONTAINS

  SUBROUTINE process_data(factors, n)
    USE data
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i

  END SUBROUTINE process_data

END MODULE prog
```

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

- MODULE prog
- SUBROUTINE process_data
- Input array factors

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
  CONTAINS

  SUBROUTINE process_data(factors, n)
    USE data
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i

  END SUBROUTINE process_data

END MODULE prog
```

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

- MODULE prog
- SUBROUTINE process_data
- Input array factors
- Length of factors: input var n

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
  CONTAINS

  SUBROUTINE process_data(factors, n)
    USE data
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i

  END SUBROUTINE process_data

END MODULE prog
```

F2PY Wrapping Example F90 Code

test_data.f90

```
MODULE data
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE data
```

test_data.f90

- MODULE data
- Data array test_arr

test_prog.f90

- MODULE prog
- SUBROUTINE process_data
- Input array factors
- Length of factors: input var n
- Updates test_arr in
MODULE data

test_prog.f90

```
MODULE prog
  IMPLICIT NONE
  CONTAINS

  SUBROUTINE process_data(factors, n)
    USE data
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i
    DO i = 1,n
      test_arr = test_arr - (-1)**i &
        * test_arr**factors(i)
    END DO
  END SUBROUTINE process_data

END MODULE prog
```

F2PY Wrapping Example: Compilation

Configuration:

- F2PY and iPython installed
- Intel Fortran under GNU/Linux

F2PY Wrapping Example: Compilation

Configuration:

- F2PY and iPython installed
- Intel Fortran under GNU/Linux

Compiling the Wrapper Module

```
$ f2py -fcompiler=intel -m testmod -c test_data.f90 test_prog.f90
```

F2PY Wrapping Example: Compilation

Configuration:

- F2PY and iPython installed
- Intel Fortran under GNU/Linux

Compiling the Wrapper Module

```
$ f2py -fcompiler=intel -m testmod -c test_data.f90 test_prog.f90
```

- `-m testmod`: Python module name is `testmod`.

F2PY Wrapping Example: Compilation

Configuration:

- F2PY and iPython installed
- Intel Fortran under GNU/Linux

Compiling the Wrapper Module

```
$ f2py -fcompiler=intel -m testmod -c test_data.f90 test_prog.f90
```

- `-m testmod`: Python module name is `testmod`.
- `-c test_data.f90...`: F90 source files to compile

F2PY Wrapping Example: Python Usage

Using the Wrapper:

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr
- [2]: test_arr initialised to zero

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr
- [2]: test_arr initialised to zero
- [3]: Assign value to test_arr

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
In [3]:testmod.data.test_arr = [1,2,3,4,5]
Out[3]:[ 1., 2., 3., 4., 5.,]
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr
- [2]: test_arr initialised to zero
- [3]: Assign value to test_arr
- [4]: From module prog call subroutine process_data

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
In [3]:testmod.data.test_arr = [1,2,3,4,5]
Out[3]:[ 1., 2., 3., 4., 5.,]
In [4]:testmod.prog.process_data([1,2])
```

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr
- [2]: test_arr initialised to zero
- [3]: Assign value to test_arr
- [4]: From module prog call subroutine process_data

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
In [3]:testmod.data.test_arr = [1,2,3,4,5]
Out[3]:[ 1., 2., 3., 4., 5.,]
In [4]:testmod.prog.process_data([1,2])
```

- [4]: Array length not explicitly passed

F2PY Wrapping Example: Python Usage

Using the Wrapper:

- [1]: Load testmod
- [2]: module data, variable test_arr
- [2]: test_arr initialised to zero
- [3]: Assign value to test_arr
- [4]: From module prog call subroutine process_data

Python Session

```
$ ipython -pylab
...
For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:[ 0., 0., 0., 0., 0.,]
In [3]:testmod.data.test_arr = [1,2,3,4,5]
Out[3]:[ 1., 2., 3., 4., 5.,]
In [4]:testmod.prog.process_data([1,2])
In [5]:testmod.data.test_arr
Out[5]:[ -2.,-12.,-30.,-56.,-90.,]
```

- [4]: Array length not explicitly passed
- [5]: Check result

Additional Remarks on F2PY

- F2PY is quite flexible
 - ▶ Examples used one-pass compilation
 - ▶ Two-step process with intermediary .pyf file
 - ▶ Uses F90 like syntax to define the interface
 - ▶ Wrap only part of a code

Additional Remarks on F2PY

- F2PY is quite flexible
 - ▶ Examples used one-pass compilation
 - ▶ Two-step process with intermediary .pyf file
 - ▶ Uses F90 like syntax to define the interface
 - ▶ Wrap only part of a code
- F2PY makes F77 code somewhat more palatable
 - ▶ Allows specification of input/output intent for F77 code
 - ▶ Hide the allocation of temporaries
 - ▶ Can specify dimension checks
 - ▶ Specified in .pyf file, runs as compiled C code

Outline

- ① Using Python with External Compiled Code
- ② Using F2PY to Wrap Fortran Code
- ③ Retroactive Application of Object Oriented Design

Wrapping to Provide Object Oriented (OO) Structure

- Python namespaces keep F90 module variables apart
 - ▶ Same-named variables in different modules don't conflict
 - ▶ Namespaces easy to manipulate
 - ▶ Python structure independent of Fortran structure

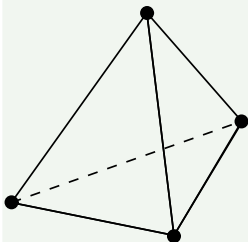
Wrapping to Provide Object Oriented (OO) Structure

- Python namespaces keep F90 module variables apart
 - ▶ Same-named variables in different modules don't conflict
 - ▶ Namespaces easy to manipulate
 - ▶ Python structure independent of Fortran structure
- Encapsulate Fortran data and routines as objects

Wrapping to Provide Object Oriented (OO) Structure

- Python namespaces keep F90 module variables apart
 - ▶ Same-named variables in different modules don't conflict
 - ▶ Namespaces easy to manipulate
 - ▶ Python structure independent of Fortran structure
- Encapsulate Fortran data and routines as objects

Simple Tetrahedral Mesh Element



- Mesh consists of several tetrahedra
- Each tetrahedron defined by four nodes
- Nodes shared by adjacent tetrahedra

Data Storage and Usage in Existing Fortran 90 Code

Fortran Data Representation

- Global list of node co-ordinates

Coordinate Representation

```
nodeCoords(1,:) = (/x1,y1,z1/),  
...  
nodeCoords(n,:) = (/xn,yn,zn/)
```

Data Storage and Usage in Existing Fortran 90 Code

Fortran Data Representation

- Global list of node co-ordinates
- Four coordinate indices per element

Coordinate Representation

```
nodeCoords(1,:) = (/x1,y1,z1/),  
...  
nodeCoords(n,:) = (/xn,yn,zn/)
```

Element Representation

```
elements(1)%nodes = (/1,7,10,11/)  
...  
elements(m)%nodes = (/11,j,k,l/)
```

Data Storage and Usage in Existing Fortran 90 Code

Fortran Data Representation

- Global list of node co-ordinates
- Four coordinate indices per element
- Elements can share nodes

Coordinate Representation

```
nodeCoords(1,:) = (/x1,y1,z1/),  
...  
nodeCoords(n,:) = (/xn,yn,zn/)
```

Element Representation

```
elements(1)%nodes = (/1,7,10,11/)  
...  
elements(m)%nodes = (/b11,j,k,l/)
```

Data Storage and Usage in Existing Fortran 90 Code

Fortran Data Representation

- Global list of node co-ordinates
- Four coordinate indices per element
- Elements can share nodes

Coordinate Representation

```
nodeCoords(1,:) = (/x1,y1,z1/),  
...  
nodeCoords(n,:) = (/xn,yn,zn/)
```

Element Representation

```
elements(1)%nodes = (/1,7,10,11/)  
...  
elements(m)%nodes = (/11,j,k,l/)
```

Using the Data in Fortran

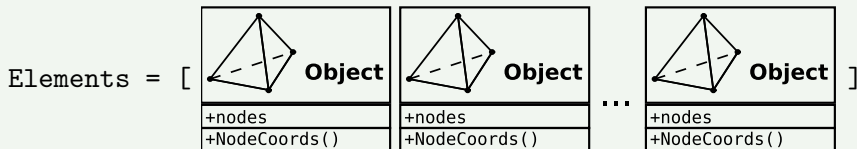
- Get the node coordinates for element 5

Fortran Code

```
nodeCoords(elements(5)%nodes,:)
```

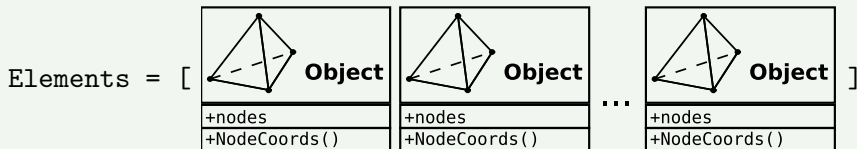
Wrapped OO Structure in Python

Python Data Representation



Wrapped OO Structure in Python

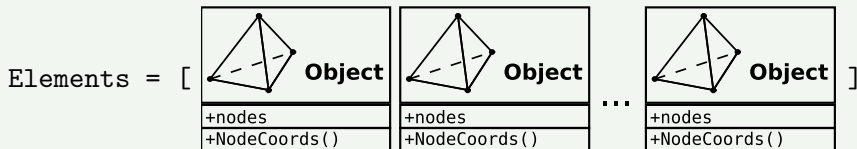
Python Data Representation



- Data stored in original F90 arrays
- Some additional Python code

Wrapped OO Structure in Python

Python Data Representation



- Data stored in original F90 arrays
- Some additional Python code

Using the Data in Python

- Get the node coordinates for element 5

Python Code

```
Elements[4].nodeCoords()
```

OO Structure Benefits

OO Python vs. Fortran

Python Code

```
Elements[4].nodeCoords()
```

Fortran Code

```
nodeCoords(elements(5).nodes)
```

- A fairly trivial example

OO Structure Benefits

OO Python vs. Fortran

Python Code

```
Elements[4].nodeCoords()
```

Fortran Code

```
nodeCoords(elements(5).nodes)
```

- A fairly trivial example
- ① Encapsulation
 - ▶ Hides implementation details
 - ▶ Reduces coupling between different parts of a program
 - ▶ Facilitates code re-use

OO Structure Benefits

OO Python vs. Fortran

Python Code

```
Elements[4].nodeCoords()
```

Fortran Code

```
nodeCoords(elements(5).nodes)
```

- A fairly trivial example

① Encapsulation

- ▶ Hides implementation details
- ▶ Reduces coupling between different parts of a program
- ▶ Facilitates code re-use

② Abstraction

- ▶ Fewer distractions
- ▶ Concentrate on problems at a higher level
- ▶ Leads to cleaner program design

Conclusion

- Wrapping existing Fortran codes with Python
- Retroactively provide OO structure
- Deal with increasing software complexity
- Avoids risks inherent in a rewrite
- Additional benefits