

f2py

Python for computational science

16 - 18 October 2017

CINECA

- f2py allows to build an extension module that interfaces Python to Fortran 77/90/95 routines
- Let's write a Fortran function to interface a simple `array_multiply` routine

```
(mult.f90)

subroutine fmult(a,b,c,n)
implicit none
real*8 :: a(*)
real*8 :: b(*)
real*8 :: c(*)
integer :: n, i
do i =1,n
  c(i) = a(i) * b(i)
enddo
end
```

```
$ f2py -c -m mult mult.f90
```

```
$ python -c 'import mult'
```

```
>>> import mult
```

```
>>> print mult.fmult.__doc__
```

```
fmult - Function signature:
```

```
  fmult(a,b,c,n)
```

```
Required arguments:
```

```
  a : input rank-1 array('d') with bounds (*)
```

```
  b : input rank-1 array('d') with bounds (*)
```

```
  c : input rank-1 array('d') with bounds (*)
```

```
  n : input int
```

```
>>> a = np.ones(10000)+ 3; c = np.empty(a.size)
```

```
>>> b = np.ones(10000)+ 1.5
```

```
>>> mult.fmult(a,b,c,a.size)
```

f2py



- One can improve the interface automatically built to make it more pythonic

```
$ f2py -h mult.pyf -m mult mult.f90
```

```
(mult.pyf)
python module mult
  interface
    subroutine fmult(a,b,c,n)
      real*8 dimension(*) :: a
      real*8 dimension(*) :: b
      real*8 dimension(*) :: c
      integer :: n
    end subroutine fmult
  end interface
end python module mult
```

- One can improve the interface automatically built

```
$ f2py -h mult.pyf -m mult mult.f90
```

```
(mult.pyf)
python module mult
  interface
    subroutine fmult(a,b,c,n)
      real*8 dimension(*) :: a
      real*8 dimension(*) :: b
      real*8 dimension(*) :: c
      integer :: n
    end subroutine fmult
  end interface
end python module mult
```

```
(mult.pyf)
python module mult
  interface
    subroutine fmult(a,b,c,n)
      real*8 dimension(n) :: a
      real*8 dimension(n) :: b
      real*8 intent(out),
dimension(n) :: c
      integer intent(hide), depend(a) ::
n=len(a)
    end subroutine fmult
  end interface
end python module mult
```

- Let us check the new interface

```
$ rm mult.so
```

```
$ f2py -c -m mult mult.pyf mult.f90
```

```
$ python -c 'import mult'
```

```
>>> import mult
```

```
>>> print mult.fmult.__doc__
```

```
fmult - Function signature:
```

```
    c = fmult(a,b)
```

```
Required arguments:
```

```
    a : input rank-1 array('d') with bounds (n)
```

```
    b : input rank-1 array('d') with bounds (n)
```

```
Return objects:
```

```
    c : rank-1 array('d') with bounds (n)
```

```
>>> import numpy as np
```

```
>>> a = np.array([1, 3, 4])
```

```
>>> b = np.array([2, 5, 1.5])
```

```
>>> c = mult.fmult(a, b)
```

```
>>> f = mult.fmult([3,4], [1.2,1.4])
```

```
>>> f
```

```
array([ 3.6,  5.6])
```

- Inserting directives in fortran source

```
subroutine fmult(a,b,c,n)
implicit none
real*8 :: a(n)
real*8 :: b(n)
real*8 :: c(n)
integer :: n, i
!f2py intent(hide), depend(a) :: n=len(a)
!f2py real*8          :: a(n)
!f2py real*8          :: b(n)
!f2py real*8, intent(out) :: c(n)
do i =1,n
    c(i) = a(i) * b(i)
enddo
end
```

```
$ f2py -c -m mult2 mult2.f90

>>> import numpy as np
>>> import mult2
>>> a = np.array([1, 3, 4])
>>> b = np.array([2, 5, 1.5])
>>> c = mult.fmult(a, b)

>>> f = mult.fmult([3,4], [1.2,1.4])
>>> f
array([ 3.6,  5.6])
```

- This fractal is defined by the iteration

$$z \leftarrow z^2 + c$$

where z and c are complex variables. This expression is iterated; if z stays finite, c belongs to the Mandelbrot set


```
subroutine single_point_mandelbrot(z,c,zout)
```

(mandel.f90)

```
! *****
```

```
! *      compute single point mandlebrot      *
```

```
! *****
```

```
implicit none
```

```
integer :: I
```

```
complex*16 :: z, c, zout
```

```
do i = 1,100
```

```
    z = z*z + c
```

```
    if ((real(z)**2 + aimag(z)**2 ) .gt. 1000.0) then
```

```
        exit
```

```
    endif
```

```
enddo
```

```
zout = z
```

```
end subroutine
```

f2py



- Let us get the new module a more pythonic interface

```
$ f2py -c -m mandel mandel.f90
```

```
$ f2py -h mandel.pyf -m mandel mandel.f90
```

(mandel.pyf)

```
! Note: the context of this file is case sensitive.
```

```
python module mandel ! in
```

```
    interface ! in :mandel
```

```
        subroutine single_point_mandelbrot(z,c,zout) ! in :mandel:mandel.f90
```

```
            complex*16 :: z
```

```
            complex*16 :: c
```

```
            complex*16 :: zout
```

```
        end subroutine single_point_mandelbrot
```

```
    end interface
```

```
end python module mandel
```

```
! This file was auto-generated with f2py (version:2).
```

f2py



- Let us get the new module a more pythonic interface

```
$ f2py -c -m mandel mandel.f90
```

```
$ f2py -h mandel.pyf -m mandel mandel.f90
```

(mandel.pyf)

```
! Note: the context of this file is case sensitive.
```

```
python module mandel ! in
```

```
    interface ! in :mandel
```

```
        subroutine single_point_mandelbrot(z,c,zout) ! in :mandel:mandel.f90
```

```
            complex*16 :: z
```

```
            complex*16 :: c
```

```
            complex*16, intent(out) :: zout
```

```
        end subroutine single_point_mandelbrot
```

```
    end interface
```

```
end python module mandel
```

```
! This file was auto-generated with f2py (version:2).
```

- Let's test the module (and the interface)

```
$ rm mandel.so
```

```
$ f2py -c -m mandel mandel.pyf mandel.f90
```

```
$ python -c 'import mandel'
```

```
>>> import mandel
```

```
>>> print mandel.single_point_mandelbrot.__doc__
```

```
single_point_mandelbrot - Function signature:
```

```
    zout = single_point_mandelbrot(z,c)
```

```
Required arguments:
```

```
    z : input complex
```

```
    c : input complex
```

```
Return objects:
```

```
    zout : complex
```

- Let's test the module (and the interface)

```
In [1]: z = complex(1,2)
```

```
In [2]: c = complex(1,2)
```

```
In [3]: zout = mandel.single_point_mandelbrot(z,c)
```

```
In [4]: zout
```

```
Out[4]: (-31-22j)
```

- Now we can employ our function
- First, we need to wrap our function

```
def myfunc(a,b):  
    return mandel.single_point_mandelbrot(a,b)
```

- And then vectorize it

```
vfunc = np.vectorize(myfunc) #now we can use  
                             #mandel.single_point_mandelbrot  
                             #as a ufunc
```

- Let's build the input points ...

```
x = np.linspace(-1.7, 0.6, 1000)  
y = np.linspace(-1.4, 1.4, 1000)  
xx, yy = np.meshgrid(x, y)
```

- ... and then get the complex numbers

```
def f(aa, bb):  
    return aa + 1j*bb
```

```
ff = f(xx, yy)    # build a 2d array of complex numbers
```

- Now we can use our new function and plot the results

```
z = vfunc(ff,ff) # just used our single_point_mandelbrot func  
import matplotlib.pyplot as plt  
plt.imshow(abs(z)**2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])  
plt.gray()  
plt.show()
```