Introduction to High Performance Scientific Computing

Autumn, 2018

Lecture 10

Imperial College
London

Prasun Ray

6 November 2018

Announcements

- HW 2 will be posted around 8pm today
- HW1 feedback will be provided around Wednesday next week (solutions will be posted at the end of this week)
- A new VM is being installed on computer in Huxley 408 and 410. Should be substantial improvement on MLC VMs
  - Will post announcement when they are ready (hopefully sometime tomorrow!)

Today

- A few comments on Fortran
- F2Py with fortran modules
- Basic computer archictecture

Notes on Fortran

- Do not attempt to guess Fortran syntax! – look it up instead
- Do not develop Fortran code in Python using f2py – get a Fortran-only code working first
- Test code (compile and run) after adding a few (4-5) lines of code. Don't write 100 lines and then test!
- Lecture 7: code structure, variable types, loops, if-then, subroutines
- Lecture 8: allocatable arrays, functions modules
- Lecture 9: Lapack, f2py
  - Note: On VMs, you may need to use "f2py3" instead of "f2py"

Notes on Fortran

- **Fortran is case *in*sensitive, the variables** a **and** A **are treated as the same variable**
- **Python is case sensitive,** a **and** A **are treated as different variables**
- **This can cause problems when using f2py. Safe approach: use all lowercase letters for variable names**
- **Slicing/indexing are a little different in Fortran and Python**
  - **Fortran: counting starts from 1, the first n elements of** A **are** A(1:n)**.**

Notes on Fortran

Debugging code

- **If the compiler is giving you a series of errors, look at the topmost error message. It will include a line number**

- **If the code runs but crashes with a segmentation fault: probably a problem with array indices (e.g. trying to access the 12th element of a** dimension(11) **array**

- **If you can't tell where the code is crashing, add print statements, e.g.** print *, 1 **--some code–** print *, 2
  When the code is run, if the 1 prints to screen but not the 2, you know where the problem is.

- **If the code runs, but gives the wrong answer, add print statements outputting values of variables, try a small problem size where you know what values the variables can take**

Notes on Fortran

- Modules consist of:
  1. Module variables
  2. Module sub-programs

- Module variables are "available" throughout the module
  - They *do not* need to be declared within sub-programs
  - They *do not* need to be provided as input/output in the sub-program header

- Module variables and module sub-programs are also "available" in any program or sub-program that *uses* the module

- A module by itself doesn't do anything
  - There should be a "main" program which uses it
  - You can compile a module by itself: gfortran -c module.f90
  - But to generate an executable, you need a ***program***:
    gfortran -o program.exe module.f90 program.f90

  When re-compiling, first remove the previous .mod file

Notes on Fortran

- **Compiling code that uses lapack routines:**

```
$ gfortran –c program.f90
$ gfortran –o program.exe program.o –llapack
```

or

```
$ gfortran –o program.exe program.f90 –llapack
```

Similarly, with f2py:

```
$ f2py –llapack –c program.f90 –m module_name
```

F2Py and fortran modules

- F2Py will recognize subroutines and functions in modules
- What about variables?
  - Try f2py with circle module from last week (*f2pymodule_circle.f90)*

```
$ f2py -c f2pymodule_circle.f90 -m cmod
```

```
In [10]: import cmod

In [11]: cmod.<tab>
cmod.circle  cmod.so
```

Need to look at *cmod.circle*

F2Py and fortran modules

Need to look at *cmod.circle*:

```
In [12]: cmod.circle?
Docstring:
'd'-scalar
initialize_pi()

Wrapper for ``initialize_pi``.

circumference = circumference(radius)

Wrapper for ``circumference``.

Parameters
----------
radius : input float

Returns
-------
circumference : float
area = area(radius)
```

And similar info for “area”

F2Py and fortran modules

How do we access variables and methods in *cmod.circle*?

```
In [23]: cmod.circle.<tab>
cmod.circle.area            cmod.circle.circumference
cmod.circle.initialize_pi   cmod.circle.pi
```

Can initialize *pi* in python:

```
In [9]: cmod.circle.pi
Out[9]: array(0.0)

In [10]: cmod.circle.pi = pi

In [11]: cmod.circle.pi
Out[11]: array(3.141592653589793)
```

F2Py and fortran modules

How do we access variables and methods in *cmod.circle*?

```
In [23]: cmod.circle.<tab>
cmod.circle.area            cmod.circle.circumference
cmod.circle.initialize_pi   cmod.circle.pi
```

Can initialize *pi* in python:

```
In [9]: cmod.circle.pi
Out[9]: array(0.0)

In [10]: cmod.circle.pi = pi

In [11]: cmod.circle.pi
Out[11]: array(3.141592653589793)
```

Can also initialize allocatable arrays, see *f2pymodule_circle.f90…*

F2Py and fortran modules

Can also initialize allocatable arrays, see *f2pymodule_circle_array.f90*:

```
!module for computing circumference, area, and "mass" of circle
module circle
    implicit none
    real(kind=8) :: pi
    real(kind=8), allocatable, dimension(:) :: weights,mass
    save

       ...
       ...

subroutine compute_mass(radius,mass)
    !compute mass = weights*area
    implicit none
    real(kind=8), intent(in) :: radius
    real(kind=8), intent(out) :: mass(:)

    mass = weights*area(radius)
end subroutine compute_mass
```

F2Py and fortran modules

Basic steps:

1. **Compile with f2py:**

```
$ f2py -c f2pymodule_circle_array.f90 -m cmoda
```

F2Py and fortran modules

Basic steps:

1. **Compile with f2py:**
```
$ f2py -c f2pymodule_circle_array.f90 -m cmoda
```

2. **Import module in python:**
```
In [4]: import cmoda
```

F2Py and fortran modules

Basic steps:

1. **Compile with f2py:**
```
$ f2py -c f2pymodule_circle_array.f90 -m cmoda
```

2. **Import module in python:**
```
In [4]: import cmoda
```

3. **Initialize variables:**

```
In [15]: cmoda.circle.pi=pi

In [16]: cmoda.circle.weights=arange(5)

In [17]: cmoda.circle.pi,cmoda.circle.weights
Out[17]: (array(3.141592653589793), array([ 0., 1., 2., 3., 4.]))
```

F2Py and fortran modules

Basic steps:

1. **Compile with f2py:**
```
$ f2py -c f2pymodule_circle_array.f90 -m cmoda
```

2. **Import module in python:**
```
In [4]: import cmoda
```

3. **Initialize variables:**

```
In [15]: cmoda.circle.pi=pi

In [16]: cmoda.circle.weights=arange(5)

In [17]: cmoda.circle.pi,cmoda.circle.weights
Out[17]: (array(3.141592653589793), array([ 0., 1., 2., 3., 4.]))
```

4. **Compute mass:**

```
In [18]: cmoda.circle.compute_mass(2.0)

In [19]: cmoda.circle.mass
Out[19]: array([  0. , 12.56637061, 25.13274123, 37.69911184,
50.26548246])
```

F2Py and fortran modules

Exercise: How does f2py treat the subroutine below with weights now provided as input? Can n ***be removed from the input?***

```
subroutine compute_mass(n,weights,radius)
    !compute mass = weights*area
    implicit none
    integer :: n
    real(kind=8), dimension(n), intent(in) :: weights
    real(kind=8), intent(in) :: radius

    if (.not. allocated(mass)) allocate(mass(size(weights)))
    mass = weights*area(radius)

end subroutine compute_mass
```

Notes on compiling

- **Up to now:**

```
$ gfortran –c program.f90
$ gfortran –o program.exe program.o –llapack
```

or

```
$ gfortran –o program.exe program.f90 –llapack
```

- **But typically want to turn on *optimization* -O flag:**

```
$ gfortran –O3 –c program.f90
```

- **-O3 is highest level of optimization (can also use -O1, -O2)**

Notes on compiling

- Look at midpoint_time.f90 compiled with and without -O3:

```
$ ./mt2.exe
 elapsed cpu time (seconds) =  0.3251049999999998
 elapsed wall time (seconds)=   0.326231003
 N=        25600
 sum=   3.1415926537169634
 error=   1.2717027431108363E-010

$ ./mt2_O3.exe
 elapsed cpu time (seconds) =  0.1437609999999997
 elapsed wall time (seconds)=   0.144617006
 N=        25600
 sum=   3.1415926537169634
 error=   1.2717027431108363E-010
```

- Optimization can make a substantial difference
- f2py uses -O3 by default

Vectorization

- **Many built-in functions in Fortran can operate on arrays**
  - ***e.g.* with an array (or matrix), x, can compute exp(x) all at once**
- **We know it is important to avoid loops in Python, does the same apply to Fortran?**

Vectorization

- **Many built-in functions in Fortran can operate on arrays**
  - ***e.g.* with an array (or matrix), x, can compute exp(x) all at once**
- **We know it is important to avoid loops in Python, does the same apply to Fortran?**

- **Consider a simple example: computing the sine of a large array, sin(x)**

- **Python:**

```
#compute sin(x) using loop
def sin_loop(x):
    s = np.zeros_like(x)
    for i,xi in enumerate(x):
        s[i] = np.sin(xi)
    return s

#vectorized
def sin_vec(x):
    s = np.sin(x)
    return s
```

Vectorization

- **Many built-in functions in Fortran can operate on arrays**
  - ***e.g.* with an array (or matrix), x, can compute exp(x) all at once**
- **We know it is important to avoid loops in Python, does the same apply to Fortran?**

- **Consider a simple example: computing the sine of a large array, sin(x)**

- **Python: sine_array.py**

```
#compute sin(x) using loop
def sin_loop(x):
    s = np.zeros_like(x)
    for i,xi in enumerate(x):
        s[i] = np.sin(xi)
    return s

#vectorized
def sin_vec(x):
    s = np.sin(x)
    return s
```

Vectorization

- **Python:**

```
In [12]: x = np.random.randn(500000)

In [13]: timeit sin_loop(x)
808 ms ± 32.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)

In [15]: timeit sin_vec(x)
11.2 ms ± 346 µs per loop (mean ± std. dev. of 7 runs, 100
loops each)
```

  - **Vectorized version is 70x faster**
  - **The increase in speed does depend on the array size**
  - **What about Fortran?**

Vectorization

- **What about Fortran? See sine_array.f90**
- **Compile module with f2py:** $ f2py -c sine_array.f90 -m s1

Import and run in python:

```
In [25]: from s1 import sine_array as sa

In [26]: sa.x = x

In [27]: n = x.size

In [28]: timeit sa.sin_loop(n)
10.5 ms ± 245 µs per loop (mean ± std. dev. of 7 runs, 100
loops each)

In [29]: timeit sa.sin_vec(n)
10.6 ms ± 192 µs per loop (mean ± std. dev. of 7 runs, 100
loops each)
```

Vectorization

- What about Fortran? See sine_array.f90
  - Vectorization is *essential* in Python (or Matlab)
  - Much less important in Fortran
    - Provided you optimize properly when compiling!