HPSC 101 — Lecture 8

This lecture:

- Fortran subroutines and functions
- Arrays
- Dynamic memory

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of modules.

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of modules.

Functions take some input arguments and return a single value.

Usage: `y = f(x)` or `z = g(x,y)`

Should be declared as external with the type of value returned:

```
real(kind=8), external ::  f
```

Fortran functions

```
1  ! $UWHPSC/codes/fortran/fcn1.f90
2
3  program fcn1
4      implicit none
5      real(kind=8) :: y,z
6      real(kind=8), external :: f
7
8      y = 2.
9      z = f(y)
10     print *, "z = ",z
11 end program fcn1
12
13 real(kind=8) function f(x)
14     implicit none
15     real(kind=8), intent(in) :: x
16     f = x**2
17 end function f
```

Prints out: `z =  4.000000000000000`

Fortran subroutines

Subroutines have arguments, each of which might be for input or output or both.

Usage: `call sub1(x,y,z,a,b)`

Can specify the intent of each argument, e.g.

```
real(kind=8), intent(in) :: x,y
real(kind=8), intent(out) :: z
real(kind=8), intent(inout) :: a,b
```

specifies that `x, y` are passed in and not modified,
`z` may not have a value coming in but will be set by `sub1`,
`a, b` are passed in and may be modified.

After this call, `z, a, b` may all have changed.

Fortran subroutines

```
1  ! $UWHPSC/codes/fortran/sub1.f90
2
3  program sub1
4      implicit none
5      real(kind=8) :: y,z
6
7      y = 2.
8      call fsub(y,z)
9      print *, "z = ",z
10 end program sub1
11
12 subroutine fsub(x,f)
13     implicit none
14     real(kind=8), intent(in) :: x
15     real(kind=8), intent(out) :: f
16     f = x**2
17 end subroutine fsub
```

Fortran subroutines

A version that takes an array as input and squares each value:

```
1  ! $UWHPSC/codes/fortran/sub2.f90
2
3  program sub2
4      implicit none
5      real(kind=8), dimension(3) :: y,z
6      integer n
7
8      y = (/2., 3., 4./)
9      n = size(y)
10     call fsub(y,n,z)
11     print *, "z = ",z
12 end program sub2
13
14 subroutine fsub(x,n,f)
15   ! compute f(x) = x**2 for all elements of the array x
16   ! of length n.
17   implicit none
18   integer, intent(in) :: n
19   real(kind=8), dimension(n), intent(in) :: x
20   real(kind=8), dimension(n), intent(out) :: f
21   f = x**2
22 end subroutine fsub
```

Array operations in Fortran

Fortran 90 supports some operations on arrays...

```
! $UWHPSC/codes/fortran/vectorops.f90
program vectorops
    implicit none
    real(kind=8), dimension(3) :: x, y

    x = (/10.,20.,30./)          ! initialize
    y = (/100.,400.,900./)

    print *, "x = "
    print *, x

    print *, "x**2 + y = "
    print *, x**2 + y            ! componentwise
```

Array operations in Fortran

```
! $UWHPSC/codes/fortran/vectorops.f90
! continued...

    print *, "x*y = "
    print *, x*y        ! = (x(1)y(1), x(2)y(2), ...)

    print *, "sqrt(y) = "
    print *, sqrt(y)                ! componentwise


    print *, "dot_product(x,y) = "
    print *, dot_product(x,y)       ! scalar product

end program vectorops
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90
program arrayops
    implicit none
    real(kind=8), dimension(3,2) :: a
    ...
    ! create a as 3x2 array:
    A = reshape((/1,2,3,4,5,6/), (/3,2/))
```

Note:

- Fortran is case insensitive: `A = a`
- Reshape fills array by columns, so

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2,3) :: b
    real(kind=8), dimension(3,3) :: c
    integer :: i

    print *, "a = "
    do i=1,3
        print *, a(i,:)     ! i'th row
        enddo

    b = transpose(a)        ! 2x3 array

    c = matmul(a,b)         ! 3x3 matrix product
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2) :: x
    real(kind=8), dimension(3) :: y

    x = (/5,6/)
    y = matmul(a,x)     ! matrix-vector product
    print *, "x = ",x
    print *, "y = ",y
```

Linear systems in Fortran

There is no equivalent of the Matlab backslash operator for solving a linear system $Ax = b$(`b = A\b`)

Must call a library subroutine to solve a system.

Later we will see how to use LAPACK routines

for this.

Note: Under the hood, Matlab calls LAPACK too!

So does NumPy.

Array storage

Rank 1 arrays have a single index, for example:

```
real(kind=8) :: x(3)
real(kind=8), dimension(3) :: x
```

are equivalent ways to define `x` with elements `x(1), x(2), x(3).`

You can also specify a different starting index:

```
real(kind=8) :: x(0:2), y(4:6), z(-2:0)
```

These are all arrays of length 3 and this would be a valid assignment:

```
y(5) = z(-2)
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[10,20,30], [40,50,60]]
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[10,20,30], [40,50,60]]
```

Or, could be stored by columns, as done in Fortran!

Multi-dimensional array storage

$$Apy = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} \qquad Afort = \begin{bmatrix} 10 & 30 & 50 \\ 20 & 40 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (2,3))
Afort = reshape((/10,20,30,40,50,60/), (/2,3/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

```
loc 3401    Apy[0,0] = 10      Afort(1,1) = 10
loc 3402    Apy[0,1] = 20      Afort(2,1) = 20
loc 3403    Apy[0,2] = 30      Afort(1,2) = 30
loc 3404    Apy[1,0] = 40      Afort(2,2) = 40
loc 3405    Apy[1,1] = 50      Afort(1,3) = 50
loc 3406    Apy[1,2] = 60      Afort(2,3) = 60
```

Aside on np.reshape

The np.reshape method can go through data in either order:

```
>>> v = linspace(10,60,6)

>>> v
array([ 10.,  20.,  30.,  40.,  50.,  60.])

>>> reshape(v, (2,3))    # order='C' by default
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])

>>> reshape(v, (2,3), order='F')
array([[ 10.,  30.,  50.],
       [ 20.,  40.,  60.]])
```

Aside on np.reshape

The np.reshape method can go through data in either order:

```
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])

>>> A.reshape(3,2)    # order='C' by default
array([[ 10.,  20.],
       [ 30.,  40.],
       [ 50.,  60.]])

>>> A.reshape((3,2),order='F')
array([[ 10.,  50.],
       [ 40.,  30.],
       [ 20.,  60.]])
```

Note: reshape can be called as function or method of A...

```
>>> reshape(A, (3,2), order='F')
```

Aside on np.flatten

The np.flatten method converts an N-dim array to a 1-dimensional one:

```
>>> A = np.array([[10.,20,30],[40,50,60]])
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])


>>> A.flatten()     # Default is 'C'
array([ 10.,  20.,  30.,  40.,  50.,  60.])


>>> A.flatten('F') # Fortran ordering
array([ 10.,  40.,  20.,  50.,  30.,  60.])
```

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

Memory allocation

```
real(kind=8) dimension(:), allocatable :: x
real(kind=8) dimension(:,:), allocatable :: a

allocate(x(10))
allocate(a(30,10))

! use arrays...

! then clean up:
deallocate(x)
deallocate(a)
```

Memory allocation

If you might run out of memory, use optional argument stat to return the status...

```
real(kind=8), dimension(:,:), allocatable :: a

allocate(a(30000,10000), stat=alloc_error)

if (alloc_error /= 0) then
    print *, "Insufficient memory"
    stop
    endif
```

Passing arrays to subroutines — code with bug

```
1  ! $CLASSHG/codes/fortran/arraypassing1.f90
2
3  program arraypassing1
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing1
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 3.
23     implicit none
24     real(kind=8), intent(inout) :: a(3)
25     integer i
26     do i = 1,3
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: `x` is a scalar, setvals dummy argument `a` is an array.

Passing arrays to subroutines

The `call setvals(x)` statement passes the address where `x` is stored.

In the subroutine, the array `a` of length 3 is assumed to start at this address. So next `3*8 = 24` bytes are assumed to be elements of `a(1:3)`.

In fact these 24 bytes are occupied by

x. (8 bytes),
y. (8 bytes),
i. (4 bytes),
j. (4 bytes).

So setting `a(1:3)` changes all these variables!

Passing arrays to subroutines

This produces:

```
x =     5.00000000000000
y =     5.00000000000000
i =    1075052544
j =             0
```

Nasty!!

- The storage location of `x` and the next 2 storage locations were all set to the floating point value 5.0e0
- This messed up the values originally stored in `y, i, j`.
- Integers are stored differently than floats. Two integers take up 8 bytes, the same as one float, so the assignment `a(3) = 5.` overwrites both `i` and `j`.
- The first half of the float 5., when interpreted as an integer, is huge.

Passing arrays to subroutines — another bug

```
1  ! $CLASSHG/codes/fortran/arraypassing2.f90
2
3  program arraypassing2
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing2
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 1000.
23     implicit none
24     real(kind=8), intent(inout) :: a(1000)
25     integer i
26     do i = 1,1000
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: We now try to set 1000 elements in memory!

Passing arrays to subroutines

This compiles fine, but running it gives:

```
Segmentation fault
```

This means that the program tried to change a value of memory it was not allowed to.

Only a small amount of memory is devoted to the variables declared.

The memory we tried to access might be where the program itself is stored, or something related to another program that's running.

Segmentation faults

Debugging segmentation faults can be difficult.

Tip: Compile using `-fbounds-check` flag of gfortran.

This catches some cases when you try to access an array out of bounds.

But not the case just shown! The variable was passed to a subroutine that doesn't know how long the array should be.

For a case where this helps, see
`$UWHPSC/codes/fortran/segfault1.f90`