This lecture:
- Fortran subroutines and functions
- Arrays
- Dynamic memory

Reading:
- class notes: Fortran Arrays
- class notes: Fortran Subroutines and Functions
- class notes: gfortran flags

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: \[ y = f(x) \quad \text{or} \quad z = g(x,y) \]

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

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

Fortran functions

```fortran
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
!
program fcn1
implicit none
real(kind=8) :: y,z
real(kind=8), external :: f

y = 2.
z = f(y)
print *, "z = ",z
end program fcn1

real(kind=8) function f(x)
implicit none
real(kind=8), intent(in) :: x
f = x**2
end function f
```

Prints out: \( z = 4.00000000000000 \)
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.

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

```
! $WHPSC/codes/fortran/sub1.f90
program sub1
  implicit none
  real(kind=8) :: y,z
  y = 2.0
  call fsub(y,z)
  print *, "z = ", z
end program sub1

 subroutine fsub(x,f)
  implicit none
  real(kind=8), intent(in) :: x
  real(kind=8), intent(out) :: f
  f = x**2
end subroutine fsub
```

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

```
! $WHPSC/codes/fortran/sub2.f90
program sub2
  implicit none
  real(kind=8), dimension(3) :: y,z
  integer n
  y = (/2.3, 3., 4.1/)
  n = size(y)
  call fsub(y,n)
  print *, "z = ", z
end program sub2

 subroutine fsub(x,n,f)
  implicit none
  real(kind=8), dimension(n), intent(in) :: x
  real(kind=8), dimension(n), intent(out) :: f
  f = x**2
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 = ", x
print *, "x**2 + y = ", x**2 + y ! componentwise

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

```fortran
! $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 = ", a
print *, "i’th row ", i
enddo

b = transpose(a) ! 2x3 array

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

Array operations in Fortran — Matrices

```fortran
real(kind=8), dimension(3,2) :: a
real(kind=8), dimension(2) :: x
real(kind=8), dimension(3) :: y

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

Linear systems in Fortran

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

Must call a library subroutine to solve a system.

Later we will see how to use LAPACK for this.

**Note:** Under the hood, Matlab calls LAPACK too!

So does NumPy.
### Array storage

**Rank 1 arrays** have a single index, for example:

```fortran
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:

```fortran
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.

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:

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

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

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

Suppose the array storage starts at memory location 3401.

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

<table>
<thead>
<tr>
<th>loc</th>
<th>Apy[0,0] = 10</th>
<th>Apy[0,1] = 20</th>
<th>Apy[0,2] = 30</th>
<th>Apy[1,0] = 40</th>
<th>Apy[1,1] = 50</th>
<th>Apy[1,2] = 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>3401</td>
<td>Afort(1,1) = 10</td>
<td>Afort(2,1) = 40</td>
<td>Afort(1,2) = 20</td>
<td>Afort(2,2) = 50</td>
<td>Afort(1,3) = 30</td>
<td>Afort(2,3) = 60</td>
</tr>
</tbody>
</table>
Aside on np.reshape

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

```python
>>> 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.]]
```

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

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

Aside on np.flatten

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

```python
>>> 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!

The good news:

Fortran 90 allows dynamic memory allocation.

Memory allocation

```fortran
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...

```fortran
real(kind=8), dimension(:,,:), allocatable :: a
allocate(a(30000,10000), stat=alloc_error)
if (alloc_error /= 0) then
  print *, "Insufficient memory"
  stop
endif
```

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 8
Passing arrays to subroutines — code with bug

```
1  ! $CLASSING/codes/fortran/arraypassing.f90
2  program arraypassing1
3    implicit none
4    real(kind=4) :: x,y
5    integer :: i,j
6    x = 1.
7    y = 2.  
8 10     i = 3
11    j = 4
12    call setvals(i)
13    print *, "x = ", x
14    print *, "y = ", y
15    print *, "i = ", i
16    print *, "j = ", j
17    end subroutine setvals
18 20    x = 5.
19    end program arraypassing1
```

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

R.J. LeVeque, University of Washington  AMath 483/583, Lecture 8

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!

R.J. LeVeque, University of Washington  AMath 483/583, Lecture 8

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) = 5.0` overwrites both `i` and `j`.
- The first half of the float 5.0, when interpreted as an integer, is huge.

R.J. LeVeque, University of Washington  AMath 483/583, Lecture 8
Passing arrays to subroutines — another bug

```
program array_passing2
  implicit none
  real(kind=4) :: x,y
  integer :: i
  x = 1; y = 1; i = 1
  call setvals(i)
  print *, y = x,y
  print *, y = x
  print *, y = x
end program array_passing2

subroutine setvals(i)
  implicit none
  real(kind=4), intent(inout) :: a(1000)
  integer :: i
  do i = 1,1000
    a(i) = i
  end do
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 option 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