Lecture II: Continuing with the basic constructs

Outline

- Arrays in Fortran
  - Array syntax
  - Dynamic memory allocation
- More about program structuring
  - Procedure types
  - About procedure arguments
  - Modules in Fortran

Fortran arrays

- *Fortran arrays* enable a natural and versatile way to access multi-dimensional data during computation
  - Matrices, vectors,...
- An array has a particular data type, same for all elements
- *Dimension* specified in the variable declaration

Array syntax

- In older Fortran, arrays were traditionally accessed element-by-element basis
- Modern Fortran has a way of accessing several elements in one go: *array syntax*

```
y(:) = y(:)  +  A(:,j) * x(j)
```

- Array syntax improves code readability and performance

Array syntax

- Element-by-element initialization

```
do j = 0, 10
   vector (j) = 0
   idx (j) = j
end do
```

- Using array syntax in initialization

```
vector = 0
! or
vector(:) = 0

idx(0:10) = (/ (j, j = 0, 10) /)
```

Array syntax

- Array syntax allows for less explicit DO loops

```
integer :: m = 100, n = 200
real :: a(m,n) , x(n), y(m)
integer :: i , j

y = 0.0
outer_loop : do j = 1, n
  inner_loop : do  i = 1, m
    y(i) = y(i) + a(i,j) * x(j)
  end do inner_loop
end do outer_loop
```

```
integer :: m = 100, n = 200
real :: a(m,n) , x(n), y(m)
integer :: j

y = 0.0
outer_loop : do j = 1, n
    y(:) = y(:) + a(:,j) * x(j)
end do outer_loop
```

Array sections

- With the array syntax we can access a part of an array in a pretty intuitive way - enter *array sections*

```
Sub_Vector ( 3:N+8)  = 0
Every_Third ( 1:3*N+1 : 3 ) = 1
Diag_Block ( i-1:i+1, j-2:j+2 ) = k
```

- Sections enable us to refer to (say) a sub-block of a matrix, or a sub-cube of a 3D array:

```
real :: a ( 1000, 1000)
integer :: pixel_3d(256, 256, 256)
a(2:500, 3:300:3) = 4.0
pixel_3d (128:150, 56:80, 1:256:8) = 32000
```

- When copying array sections, then both left and right hand sides of the assignment statement has to have conforming dimensions

Dynamic memory allocation

- Memory allocation is *static* if the array dimensions have been declared at compile time
- If the sizes of an array depends on the input to program, its memory should be *allocated* at runtime
  - memory allocation becomes *dynamic*

Dynamic memory allocation

- Fortran provides two different mechanisms to allocate memory dynamically through arrays:
  - Array variable declaration has an ALLOCATABLE attribute
    - memory is allocated through the ALLOCATE statement
    - and freed through DEALLOCATE
  - A variable, which is declared in the procedure with size information coming from the argument list or from a module, is an *automatic array*
    - no ALLOCATE or DEALLOCATE is needed

Dynamic memory allocation

```
integer :: m=100, n=200, alloc_stat
integer, allocatable :: idx(:)
real, allocatable :: mat(:,:)

allocate (idx(0:m-1), stat=alloc_stat)
if (alloc_stat /= 0) call abort()

allocate (mat(m,n), stat=alloc_stat)
if (alloc_stat /= 0) call abort()
...
deallocate (idx , mat)
```

A non-zero value is written to alloc_stat in case of failure (e.g. not enough memory to allocate an array)

Memory allocation with automatic arrays

```
subroutine calculate(m, n)
  integer :: m, n ! intended dimensions
  integer :: idx(0:m-1)       ! an automatic array
  real :: mat(m,n)            ! an automatic array

! no explicit allocate – but no checks upon failure either
  ...
  call do_something(m, n, idx, mat)
  ...
! no explicit deallocate - memory gets reclaimed automatically

end subroutine calculate
```

Array intrinsic functions

- Built-in functions can apply various operations on whole array, not just array elements
- As a result either another array or just a scalar value is returned
- A subset selection through *masking* is possible
  - Masking and use of array (intrinsic) functions is often accompanied with use of FORALL and WHERE array statements

Array intrinsic functions

- `SIZE(array [,dim])` returns the number of elements in the array
- `SHAPE(array)` returns an integer vector containing the size of the array with respect to each of its dimension
- `COUNT(L_array [,dim])` returns the count of elements which are .TRUE. in the LOGICAL L_array
- `SUM(array[, dim][, mask])` returns the sum of the elements

Array intrinsic functions

- `ANY(L_array [, dim])` returns a scalar value of .TRUE. if any value in LOGICAL L_array is .TRUE.
- `ALL(L_array [, dim])` returns a scalar value of .TRUE. if all values in LOGICAL L_array are .TRUE.
- `MINVAL /MAXVAL (array [,dim] [, mask])` return the minimum/maximum value in a given array
- `MINLOC/MAXLOC (array [, mask])` return a vector of location(s) where the minimum/maximum value(s) are found

Array intrinsic functions

- `RESHAPE(array,shape)` returns a reconstructed array with different shape than in the input array, for example:
  - Can be used as a single line statement to initialize an array (often in expense of readability)
  - Create from an M-by-N matrix a vector of length MxN

```
INTEGER :: M, N
REAL :: A(M, N), V(M*N)
! Convert A into V without loops
V = RESHAPE(A, SHAPE(V))
```

Array intrinsic functions

- Some array functions manipulate vectors/matrices effectively :
  - `DOT_PRODUCT(v,w)` returns a dot product of two vectors
  - `MATMUL(A,B)` returns matrix multiply of two matrices
  - `TRANSPOSE(A)` returns transposed of the input matrix

Array intrinsic functions

- Array control statements FORALL and WHERE are commonly used in the context of manipulating arrays
- They can provide a masked assignment of values using effective vector operations

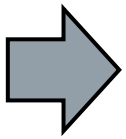
```
integer :: j, ix(5)
ix(:) = (/ (j, j=1,size(ix)) /)
where (ix == 0) ix = -9999
where (ix < 0)
    ix = -ix
elsewhere
    ix = 0
end where
```

```
integer :: j
real :: a(100,100), b(100), c(100)
! fill in diagonal matrix
forall (j=1:100) a(j,j) = b(j)
! fill in lower bi-diagonal matrix
forall (j=2:100) a(j,j-1) = c(j)
```

Loop order in multi-dimensional arrays

- Always increment the *left-most* index of multi-dimensional arrays in the *innermost* loop (i.e. fastest)
- Some compilers (with sufficient optimization flags) may re-order loops automatically

```
do i=1,N
  do j=1,M
    y(i) = y(i)+ a(i,j)*x(j)
  end do
end do
```



```
do j=1,M
  do i=1,N
    y(i) = y(i)+ a(i,j)*x(j)
  end do
end do
```

Interim summary

- *Arrays* make Fortran language a very versatile vehicle for computationally intensive program development
- Using its *array syntax*, vectors and matrices can be initialized and used in a very intuitive way
- *Dynamic memory allocation* enables sizing of arrays according to particular needs
- *Array intrinsic functions* further simplify coding effort and improve code readability

Procedure types

- There are four procedure types in Fortran: *intrinsic*, *external*, *internal* and *module* procedures
- Procedure types differ in
  - Scoping (what data and other procedures a procedure can access) and interface type (explicit or implicit)
- In Fortran the procedure arguments are always passed by reference, i.e. just as a pointer to a location in memory
- Compiler can check the argument types of the at compile time only if the interface is explicit

Procedure types, cont.

- The interfaces of the intrinsic, internal and module procedures are explicit
- The interfaces of the external procedures, such as many library subroutines, are implicit. You can write an explicit interface to those, though.
- Intrinsic procedures are the procedures defined by the programming language itself, such as

```
INTRINSIC SIN
```

Internal procedures

- Each program unit (program/subroutine/function) may contain *internal procedures*

```
subroutine mysubroutine
    …
    call myinternalsubroutine
    ...
contains
    subroutine myinternalsubroutine
        …
    end subroutine myinternalsubroutine
end subroutine mysubroutine
```

Internal procedures, cont.

- Declared at the end of a program unit after the CONTAINS statement
  - Nested CONTAINS statements are not allowed
- Scoping: internal procedure can access the parent program unit's variables and objects
- Often used for "small and local, convenience" subroutines within a program unit

External procedures

- Declared in a separate program unit
  - Referred to with the EXTERNAL keyword
  - Compiled separately and linked to the final executable
- Remember that module procedures provide much better compile time error checking
- External procedures are needed only for
  - procedures written with different programming language
  - library routines (e.g. BLAS & MPI libraries)
  - old F77 subroutines

More about procedure arguments

- Two (three) ways to pass arrays to procedures
  - Explicit shape array (dimensions passed explicitly, F77'tish)

```
subroutine foo(size1, size2, ..., matrix, ...)
    implicit none
    integer :: size1, size2
    real, dimension(size1,size2) :: matrix
    …
```

  - Assumed shape array (requires explicit interface)

```
subroutine foo(matrix)
   real, dimension(:,:) :: matrix
```

    - One can use the intrinsic function SIZE for checking the actual dimensions

More about procedure arguments

- We may pass into procedures also *other procedures* (i.e., not only data)
- Internal procedures cannot be used as arguments

```
program degtest
   implicit none
   intrinsic asin, acos, atan
   write (*,*) 'arcsin(0.5): ', deg(asin,0.5)
   write (*,*) 'arccos(0.5): ', deg(acos,0.5)
   write (*,*) 'arctan(1.0): ', deg(atan,1.0)
contains
 real function deg(f, x)
   implicit none
   intrinsic atan
   real, external :: f
   real, intent(in) :: x
   deg = 45*f(x)/atan(1.0)
 end function deg
end program degtest
```

Modular programming

- Modularity means dividing a program into minimally dependent *modules*
  - Split the program into smaller self-contained units
- Where to employ Fortran modules
  - Global definitions of procedures, variables and constants
  - Compilation-time *error checking*
  - Hiding *implementation details*
  - *Grouping* routines and data structures
  - Defining *generic procedures* and custom operators

Module procedures & variables

Declaration

```
module check
 implicit none
 integer, parameter :: &
   longint = selected_int_kind(8)
contains
  function check_this(x) result(z)
   integer(longint):: x, z
   ...
  end function
end module check
```

Module procedures are declared after the CONTAINS statement

Usage

```
program testprog
  use check
  implicit none
  integer(kind=longint) :: x,test
  test=check_this(x)
end program testprog
```

Procedures defined in modules can be referred to in any other program unit with the USE clause

- A good habit

```
use check, only: longint
```

Global data/variables

- Global variables can be accessed (read and written) from any program unit
- Fortran module variables provide controllable way to define and use global variables

```
module commons
  integer, parameter :: r = 0.42
  integer, save :: n, ntot
  real, save :: abstol, reltol
end module commons
```

  - Explicit interface: type checking, limited scope

Visibility of module objects

- Variables and procedures in modules can be PRIVATE or PUBLIC
  - PUBLIC = visible for all program units using the module (default)
  - PRIVATE will hide the objects from other program units

```
real :: x, y
private :: x
public :: y
```

Interface definition

- The procedure interfaces needs sometimes to be defined explicitly
  - Enables compilation time error checking for *external* procedures
  - Defining sc. generic procedures and in operator overloading
  - When passing user-written procedures as procedure argument

```
subroutine nag_rand(table)
 interface
   subroutine g05faf(a, b, n, x)
     real, intent(in) :: a
     real, intent(in) :: b
     integer, intent(in) :: n
     real, intent(out), dimension(n) :: x
   end subroutine g05faf
 end interface
 real, dimension(:), intent(out) :: table
 call g05faf(-1.0, 1.0, size(table), table)
end subroutine nag_rand
```

Defining an interface for the g05faf subroutine of the NAG library (generates a set of random numbers)

Summary of the latter part

- Procedural programming makes the code more readable and easier to develop
  - Procedures encapsulate some piece of work that makes sense and may be worth re-using elsewhere
  - Fortran uses *functions* and *subroutines*
  - Values of procedure arguments may be changed upon calling the procedure
- Fortran *modules* are used for grouping procedures and for data encapsulation

Programming assignment

Differential evolution is an optimization strategy for very complex unrestricted optimization problems, see http://en.wikipedia.org/wiki/Differential_evolution

The file de_opt.f95 contains Fortran modules that provide an implementation of the differential evolution scheme. This program is from the CSC Fortran 95/2003 book pg. 263-269 (with some restructuring).

Get acquainted with the code such that you can explain what happens in it. Write a main program that employs those modules for finding the minimum of a highly complex multidimensional function of your choice. Return the program source code together with sample output by the next lecture.