

# Loops and Arrays

22.901 Introduction to Computer Programming for Nuclear Engineers

January 24, 2012

# Outline

- Control Constructs
- Arrays

# Named IF Statements

- An if statement can be preceded by a name
- If this is used, the end if must be followed by the same name
- These are known as labels

```
xcheck:  if (x < 0.0) then  
    x = a  
else  
    x = c  
end if xcheck
```

# Select Case Construct

- It is very common that there may be a lot of if statements for one variable
- Instead of putting a bunch of else if constructs, use **select case** construct

```
print *, 'Happy Birthday, what is your age?'
read *,age
select case(age)
case(18)
  print *, 'You can now vote'
case(40)
  print *, 'Life begins'
case(60)
  print *, 'Free prescriptions'
case default
  print *, 'It's just another birthday!'
end select
```

# The DO Loop

- A loop with some iteration counter, has the syntax:

```
[loop name] do [loop control]
    execution statements
end do [loop name]
```

- loop name and loop control are optional
- If there is no loop control, it will go indefinitely (not good!)

# Loop Control

- Simple Loop Control:  $\langle \text{ivar} \rangle = \langle \text{LB} \rangle, \langle \text{UB} \rangle$
- $\langle \text{ivar} \rangle$  **must** be an integer variable
- $\langle \text{LB} \rangle$  and  $\langle \text{UB} \rangle$  are the lower and upper bounds respectively
- **Note:** The bounds can hold any integer expression

## The flow of a loop::

The variable starts at the lower bound

If it exceeds the upper bound, the loop exits

The loop body is executed

The variable is incremented by **ONE**

The loop starts again

# Specifying an Increment

- The default increment step is +1 (will not decrement)
- To specify an increment: `<ivar> = <lb>,<ub>,<step>`
- Therefore, to perform a decreasing loop counter:

```
history:  do i = 20,1,-2  
          execution statements  
end do history
```

# Rules of DO Loops

- Can't change bounds of loop once inside the loop
- **Never** change the value of the loop variable
- Feel free to use the loop variables value anywhere **inside** the loop
- Loop variable is not defined outside the context of a loop
- Do not use GOTO to exit a loop use EXIT

exit will break out of the nearest loop

cycle will go increment the counter and go back to the top

**Don't confuse cycle with continue**



# While Loops

- Useful when the a logical expression determines when the loop ends
- Has the syntax:

```
while (<logical expression>)  
    execution statements  
end while
```

- Loop will execute until <logical expression> is `.false.`
- I generally do not use these because they could go forever
- If a logical expression should terminate a loop I specify a do loop with a finite number of iterations
- I then use a conditional statement to exit the loop early

# Nonlinear Bisection Algorithm I

```
program nonlinear

implicit none

real(8) :: x    ! solution variable
real(8) :: xl   ! lower bound
real(8) :: xr   ! upper bound
real(8) :: f    ! final residual
real(8) :: tol  ! tolerance

! function declaration
external myfun

! ask user for guess
print *, 'Enter guess for solution of supplied function:'
read *, x

! ask for bounds
```

# Nonlinear Bisection Algorithm II

```
print *, 'Enter lower and upper bound of search:'
read *, xl, xr

! ask for tolerance
print *, 'Enter solution tolerance:'
read *, tol

! solve by bisection
call bisect(myfun, xl, xr, x, f, tol)

! print result
print *, 'Solution is:', x, ' with final residual:', f

! terminate the program
stop

end program nonlinear
```

# Nonlinear Bisection Algorithm III

```
function myfun(x)

implicit none

! formal variables
real(8) :: myfun    ! the function declaration , residual
real(8) :: x        ! the independent variable

myfun = x**2 - 4

end function myfun
```

# Nonlinear Bisection Algorithm IV

```
subroutine bisect(fun,xl,xr,x,f,tol)

implicit none

! arguments
real(8),intent(inout) :: xl      ! left bound
real(8),intent(inout) :: xr      ! right bound
real(8),intent(inout) :: x       ! result
real(8),intent(out)   :: f       ! residual
real(8),intent(in)    :: tol     ! residual tolerance

! function argument
real(8) :: fun
external fun

! local variables
real(8) :: fl      ! residual for left bound
real(8) :: fr      ! residual for right bound
```

# Nonlinear Bisection Algorithm V

```
integer :: i           ! loop counter

! determine residual bounds
fl = fun(xl)
fr = fun(xr)

! begin loop
do i = 1,100

    ! get midpoint
    x = 0.5*_8*(xl + xr)

    ! evaluate residual at midpoint
    f = fun(x)

    ! check for convergence
    if (abs(f) < tol) exit

    ! reset the bounds
```

# Nonlinear Bisection Algorithm VI

```
if (f*fl < dble(0.0)) then

    ! move right bound info to mid
    xr = x
    fr = f

else

    ! move left bound info to mid
    xl = x
    fl = f

end if

! print out information
print *, 'Iteration:', i, ' Residual:', f

end do
```

# Nonlinear Bisection Algorithm VII

```
end subroutine bisect
```



# Declaring an Array - Terminology

```
real, dimension(10) :: a  
real :: a(10),b(0:9,5,6:10)
```

- The *rank* is the number of dimensions
- a has rank 1 and b has rank 3
- The *bounds* are the upper in lower limits
- If the range is not specified default is 1:dim
- a has bounds 1:10 and b has bounds 0:9,1:5,6:10
- A dimension's *extent* is  $UB - LB + 1$
- a has extent 10 and b has extents 10,5,5

# Array Element Assignments

- An array index can be any integer expression
- In order to declare size with a variable here, must use parameter
- Later in this lecture we will talk about *allocation*

```
integer,parameter :: n=50
integer,dimension(-50:50) :: mark
integer :: i do i = -n,n
  mark(i) = 2*i
end do
```

Sets mark to -100,-98,...,90,100

# Array Element-wise Operations

- Most built-in operators/functions are elemental
- They act element-by-element on arrays

```
real,dimension(1:200) :: arr1,arr2,arr3
execution statements
arr1 = arr2 + 1.23*exp(arr3/4.56)
```

- Comparisons and logical operations are also element-wise

```
real,dimension(1:200) :: arr1,arr2,arr3
logical,dimension(1:200) :: flags
execution statements
flags = (arr1 > exp(arr2) .or. arr3 < 0.0)
```

# Array Intrinsic functions

- There are a lot of useful intrinsic procedures for arrays:
  - `size(x[,n])` - the size of the nth dimension of an array
  - `lbound(x[,n])` - the lower bound of the nth dimension of x
  - `ubound(x[,n])` - the upper bound of the nth dimension of x
  - `minval(x)` - the minimum of all elements of x
  - `maxval(x)` - the maximum of all elements of x
  - `sum(x[,n])` - the sum of all elements of x
  - `product(x[,n])` - the product of all elements of x
  - `transpose(x)` - the transpose of a matrix
  - `dot_product(x,y)` - the dot product of x and y
  - `matmul(x,y)` - 1- and 2-D matrix multiplication
- If you don't specify n and there is more than 1 dimension either, the operation is performed for all dimensions to give a scalar, or over each dimension resulting in a vector

# Element Ordering in Memory

- The traditional term is “column-major order”
- However, memory is only linear
- The easy way to remember is **first index varies the fastest**
- Also means, the first index should be the inner-most nested loop (faster code!!)
- The elements are in the following order:

$A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3)$

**Note: C, Matlab, etc. are the opposite**

# Simple I/O of Arrays

- Arrays can be included in I/O
- Remember *column-major order*

```
real, dimension(3,2) :: oxo  
read *,oxo
```

- This is exactly equivalent to:

```
real :: oxo(3,2)  
read *, oxo(1,1), oxo(2,1), oxo(3,1), oxo(1,2),  
oxo(2,2), oxo(3,2)
```

# Array Constructors

- An array constructor creates a temporary array and copies it to a variable

```
integer :: marks(1:6)  
marks = (/10,25,32,50,54,60/)
```

- Can use variable expressions:  
(/x,2.0\*y,sin(t\*w/3.0),etc./)

- A list can be set using an **Implied-DO Loop**:

- Let  $n = 9$ :            (/0.0,(k/10.0,k=1,n),1.0/)

- This lists 0.0 - 1.0 in increments of 0.1

# Allocatable Arrays

- So far when we have created arrays, we have used **static allocation**
- Allocatable arrays use **dynamic allocation**
- A new attribute, `allocatable`, is used to declare an array with *unknown* shape but *known* rank!  

```
integer, dimension(:), allocatable :: counts  
real, dimension(:,:,:), allocatable :: values
```
- They become defined when the space is allocated:  

```
allocate(counts(1:1000))  
allocate(value(0:N,-5:5,M:2*N+1))
```
- When finished using, **must** deallocate:  

```
deallocate(counts)  
deallocate(values)
```



# Cholesky Decomposition

To solve  $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ ,  $\mathbf{A}$  must be symmetric and positive definite:

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}$$

$$\forall i > j, L_{ij} = \left( A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right) / L_{jj}$$

# Cholesky Algorithm I

```
program cholesky
implicit none

integer          :: n      ! number of rows/cols in
matrix
integer          :: i      ! loop counter
real, allocatable :: A(:, :) ! input matrix

! ask user for dimensions of matrix
print *, 'Enter number of rows/cols in matrix (must be
square) '
read *, n

! allocate A and L
allocate(A(n,n))

! have user enter matrix
```

# Cholesky Algorithm II

```
print *, 'Enter matrix in column-major order'
read *, A

! perform cholesky decomposition
call cholesky_sub(A,n)

! print matrix to user
print *, 'Result::'
do i = 1,n
    print *, A(i,:)
end do

! terminate the program
stop

end program cholesky

subroutine cholesky_sub(A,n)
```

# Cholesky Algorithm III

```
implicit none

! formal vars
integer :: n      ! number of rows/cols in matrix
real    :: A(n,n) ! matrix to be decomposed

! local vars
integer :: j      ! iteration counter

! begin loop
chol: do j = 1,n

    ! perform diagonal component
    A(j,j) = sqrt(A(j,j) - dot_product(A(j,1:j-1),A(j,1:j-1))
    ))

    ! perform off-diagonal component
    if (j < n) A(j+1:n,j) = (A(j+1:n,j) - matmul(A(j+1:n,1:j-1),A(j,1:j-1))) / &
```

# Cholesky Algorithm IV

```
      &          A(j , j)  
    end do chol  
end subroutine cholesky_sub
```

# End of Class/External Assignment

To solve  $C = A * B$

$$c_{ij} = \sum_{k=1}^m a_{ik} * b_{kj}$$

A has dimensions n,m

B has dimensions p,q

Have user enter these and make sure you allocate

Check that  $m = p$ , if not terminate the code with a message

Call subroutine for matrixmultiplication

In subroutine, you will need a two do loops that are nested (one for column and one for row)

You will need intrinsic function sum

Fun this in standalone and use it in cholesky decomposition instead of matmul