

Data Analysis in Geophysics

ESCI 7205

Bob Smalley

Room 103 in 3892 (long building), x-4929

Tu/Th - 13:00-14:30

CERI MAC (or STUDENT) LAB

Lab - 24, 11/19/13

I don't know what the programming language of the year 2000 will look like, but I know it will be called Fortran.

Charles Anthony Richard Hoare

One other rather dangerous DO form is allowed under Fortran 90.

```
x=1
do x=2*x+1
if (x.gt.200) exit
if (x.lt.100) cycle print *, ' x.ge.100'
end do
print * , ' x = ',x
```

This will continue to loop until an EXIT, or GOTO statement within the loop forces the looping to end.

Place were goto sort of makes sense – want different behavior depending on how leave the loop.

```
do 100 it=1,itmax
  x0=x
  fx=f(x0,dfdx)
  dx = -fx/dfdx
  x=x0+dx
  write (*,2000)it,x,fx,dx
  if(abs(x-x0).lt.eps*abs(x)) go to 200
100 continue
  print *, 'Iteration failed to converge'
      Better do something else here!
200 x=xin
```

If you belong to the school of thought, that GOTOs should be avoided at all cost, then the Fortran CYCLE and EXIT statements are for you.

Fix the previous code to be “modular” and not use the GOTO.

Modern way.

Need something to tell you how you left the loop.
Use some sort of logical “flag”.

```
converge_flag=.false.  
do 100 it=1,itmax  
  ...fortran code from last slide...  
    if(abs(x-xo).lt.eps*abs(x)) then  
      converge_flag=.true.  
      exit  
    endif  
  If(.not. converge_flag) then  
    print *, 'Iteration failed to converge'  
    Better do something else here!  
  end if  
x=xin
```

Random stuff for loops

- Can't modify loop control variable inside loop.

Compiler will catch this.

- Can't depend on value of loop counter when exit loop by completing it (if you are lucky it will have the first value that failed the test, causing the loop to terminate).

Compiler will not catch this.

Random stuff for loops

- Traditional loops can share a labeled “end”

```
do 100 i=1,10
```

“fortran statements”

```
Do 100 j=20,30
```

“fortran statements”

```
100 continue or “executable fortran statement”
```

what are the values of *i* and *j* here?

do...enddo have to be individually paired.

In nested loops, which loop do the `exit` and `cycle` commands apply to?

-> The nest level being executed.

What if you want get all the way out of the loop?
(higher the the next level up in the nest.)

New with Fortran 90 – you can name the loops
(at beginning and end)

Refer to the loop name you want to **exit** or
cycle

```
outer: do i=1,5
middle: do j=11,25
inner: do k=21,25
        print *, i, j, k
        if(j==12)exit outer
end do inner
end do middle
end do outer
```

Implied loops

```
x = (/ (2*i, i=1,4) /)
```

Equivalent to

```
do i = 1,4  
  x(i) = 2*i  
end do
```

Implicit loops are used for initialization of arrays, reading and printing (heaviest use).

Implied loops

```
x = (/ ((i*j, i=1,4), j=1,6) /)
```

Equivalent to

```
do j = 1,6  
do i = 1,4  
x((j-1)*4+i) = i*j  
end do  
end do
```

```
k = 1  
do j = 1,6  
do i = 1,4  
x(k) = i*j  
k = k+1  
end do  
end do
```

Implied loops

Can be very handy.

```
write(*,*)((i,"*",j,"=",i*j,j=1,9),i=1,9)
```

Prints multiplication table (but all on one line).

Implied loops

Most commonly used to read and write

```
read(*,*) n, (x(i), i=1, n)
```

or

```
read(*,*) n
```

```
read(*,*) (x(i), i=1, n)
```

or

```
read(*,*) n
```

```
do i=1, n
```

```
    read(*,*) x(i)
```

```
enddo
```

All 3 seem to do the same thing – and they do – sometimes!

Say I have 3 files with the following contents

5 1 4 9 16 25

The first piece of code on the last page will read all 3 files.

5
1 8 27 64 125

The second will read the last two files.

5
1
2
3
4
5

The third will read only the last file.

This is because each read starts a new “record”=line from the input file, but if it is not done reading when it gets to the end of the line, it keeps going with the next line.

Reading and writing from files

First have to open a file and tell the program how to identify it, do this with a “unit number” (standard-in is unit 5, but can use *).

Then replace the first * in ()’s after read with “unit” number, stuff in grey is optional

```
open(unit=1, file='f3.dat')
read(unit=1, *) n, (x(i), i=1, n)
write(*, *) (x(i), i=1, n)
End
```

Can do same with output

Reading and writing from files

There are lots more options for the open
look them up.

Handling errors/end of file on input. Modern way.

```
read(1,*,iostat=iostat) n
if(iostat>0)then
print *,'something wrong'
elseif(iostat<0)then
print *,'eof'
else
write(*,*)'n=',n
endif
close(1)
```

Handling errors/end of file on input.
Old “goto” way.

```
read(1, *, end=100, err=101) n  
write(*, *) 'n=' , n  
close(1)
```

...

Somewhere else in the code

```
100 print *, 'eof'  
stop  
101 print *, 'bad input'  
stop
```

So far we've only been reading and writing what is called list-directed (free) format

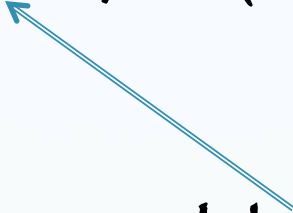
As long as input is composed of numbers it is easy (although you have to make sure you don't try to read a real into an integer), separating the numbers by spaces or commas.

A bit more complicated to read in character strings since they can have spaces (and don't have to be in quotes).

Formatted I/O

Gives more control over what it read and written.

```
WRITE(unit, "(A,F10.3,A)") "flux =" &  
      ,source_flux, " Jansky"
```



(Can use variable to specify unit number to print)

Then print a character string (A) of unspecified length

Followed by a floating point number using a total of 10 spaces, with three digits after the decimal point (sign and decimal point count),

And finally another character string.

Formatted I/O

Can also use with print and accept.

```
print    ' ("A,F10.3,A") ' "flux =" &  
        ,source_flux, " Jansky"
```

Have various ways to specify different types of numbers

I ~ integer

(In is n digits/spaces w/o leading zeros, or In.m prints at least $m \leq n$ digits, so uses leading zeros)

F ~ floating point

(Fn.m is n total digits, m after decimal point, remember to count sign and decimal point characters)

A ~ character

(An is n total characters, including spaces)

E – scientific notation

($ESn.mEo$ is n total digits, m after decimal point, normalized with 1 digit before decimal point and optionally specify 0 digits for exponent.)

($ENn.mEo$ is n total digits, m after decimal point, normalized such that exponent is multiple of 3.)

Also have

Generalized exponent (G)
Hex, Octal and Binary (H, O, B)
Logical (L).

Space (X)
Tab (T)
New line (/)

Repetition (for all format specifiers)

`rX, rFn.m`

Repeats the specification that number of times.

Formatted I/O
Old way.

Could do same

```
WRITE(unit, "(A,F10.3,A)") "flux =" &  
    ,source_flux, " Jansky"
```

Or use labels (may not work in F90...)

```
WRITE(unit,100) "flux =" &  
    ,source_flux, " Jansky"  
100 format(A,F10.3,A)
```

And share format between different statements.

Internal I/O

Read from/write to character variable.

And how to declare characters.

Can use ' or " in pairs to define strings.

```
character(len=100) :: string
```

```
character(100) :: otherstring
```


```
Character(*) :: prompt = 'enter real' !compiler will figure out length
```

```
character otherstring2(100) !this is array of 100 single characters, not 100 character long string as above
```

```
read(*, '(A)')string
```

```
read(string(10:15), *) somereal
```

Reads from character substring in positions 10 to 15 inclusive.



Q format specifier

Get length of input record.

```
character(50) buf; character(60) string; integer(4) nbytes
```

```
buf = 'this string is 29 bytes long.'
```

```
read( buf, fmt='(q)' ) nbytes
```

```
write( *,* ) nbytes
```

```
read(*,fmt='(q,a)') nbytes, string
```

```
write( *, '(i,x,3a,/,a,/,3a)' ) nbytes, '"',string,'"',' vs
```

```
','"',string(1:nbytes),'"'
```

```
end
```

Run it

```
50      nbytes equals 50 because the buffer buf is 50 bytes long.
```

```
how long is this string?
```

```
24 "how long is this string?"
```

```
nbytes now equal the number of characters entered by the user.
```

```
vs
```

```
"how long is this string?"
```

Unformatted (binary) I/O

Omit the format specifier.
Writes data in binary format.

Must be read back with exactly the same types of variables, on the same computer architecture, if you want it to make sense!!

(E.g. with the same endianness and word size.)

Example:

```
write( unit=9 ) x, y, z  
read( unit=9 ) a, b, c
```

Character strings in Fortran and C.

Can't find a limit for character string length in Fortran90

It is 255 characters in Fortran-Fortran77 (first byte is length, can only count up to 255, from 000, with 8 bits)

C character strings are “zero” terminated. They start at a given memory location and continue until a byte equal to zero is encountered.

This makes passing strings between the two languages tricky.

Arrays

Fortran also has arrays (similar to Matlab – including how elements are ordered in memory.)

You have to declare variables as being arrays.

Traditionally you had to declare the size of an array at the time you wrote the program (Fortran before Fortran95 did not have dynamic memory allocation – except on the DEC VAX, whose extended Fortran had it in the early to mid 80's, but it was not “portable”).

So far all variables have been static variables

They have a fix memory requirement, which is specified when the variable is declared.

Static arrays in particular are declared with a specified shape and extent which cannot change while a program is running (at least not in the main program).

So far all variables have been static variables

This means that when processing variable amounts of data you have to:

Dimension arrays to the largest possible size that will be required,

or

Change the sizes in the source, and re-compile every time you run (or the array is too small).

Declaring arrays with dynamic memory allocation
(size not fixed at coding time).

Having to declare arrays at compile time has long
been a complaint against Fortran

(especially by C programmers as C allowed it).

Fortran 90 (and newer) has “fixed” the problem.

Various ways to declare using allocatable option

General form

```
type, ALLOCATABLE [,attribute] :: name
```

examples

```
INTEGER, DIMENSION(:), ALLOCATABLE :: a !rank 1
```

```
INTEGER, ALLOCATABLE :: b(:, :) !rank 2
```

```
REAL, DIMENSION(:, :), ALLOCATABLE :: c !rank 2
```

Declare array using allocatable option

```
program dynamem
implicit none
integer :: nmax, istat, error
real*8, allocatable :: array(:)
...
end
```

When you need the array, use allocate.

General form

```
ALLOCATE ( name(bounds) [,STAT] )
```

Examples

```
!Some part of the code determines nmax  
nmax=100  
allocate(array1d(nmax),stat=error)  
if (stat.ne.0) then  
print*, 'error: couldnt allocate memory &  
        for array1d, nmax=',nmax  
stop  
endif
```

When done with array, should (have to!) deallocate.

General form

```
DEALLOCATE ( name [ ,STAT ] )
```

examples

```
DEALLOCATE ( a, b )  
DEALLOCATE ( c, STAT=test )  
IF (test .NE. 0) THEN  
STOP 'deallocation error'  
ENDIF
```

Status of allocatable memory

Allocated – has associated memory

Not currently allocated – no memory associated

General form

```
Allocated( name )
```

Returns `.true.` or `.false.`

Note that *you still need to know the size of the array when you declare it dynamically.*

You can compute it in your program, etc.

(this is also true in C)

You cannot just keep adding elements and have the array grow as in Matlab.

Aside

Logical variables

Define as logical

```
logical :: TF
```

They take the values

```
.true. or .false.
```

(can do stuff like `do while (.true.)`)

Status of allocatable memory

Examples

if you need memory can check and allocate if not already allocated

```
IF ( .NOT. ALLOCATED( x ) ) ALLOCATE( x( 1:10 ) )
```

Or if you don't need it, can check if allocated and get rid of it if it is allocated.

```
IF ( ALLOCATED( x ) ) DEALLOCATE( x )
```

Memory leaks

Normally, the program takes responsibility for allocating and deallocating storage to (static) variables. When using dynamic memory allocation, however, this responsibility falls to the programmer.

Storage allocated through the `ALLOCATE` statement may only be recovered by: a corresponding `DEALLOCATE` statement, or the program terminating.

Memory leaks

Storage allocated to local variables (in say a subroutine or function) must be deallocated before exiting the procedure.

When leaving a procedure all local variables are deleted from memory and the program releases any associated storage for use elsewhere,

Memory leaks

HOWEVER any storage allocated through the `ALLOCATE` statement will remain 'in use' even though it has no associated variable name!

Storage allocated, but no longer accessible, cannot be released or used elsewhere in the program and is said to be in an 'undefined' state.

This reduction in the total storage available to the program called is a "memory leak".

Memory leaks

And to make matters worse –

memory leaks are cumulative, repeated use of a procedure which contains a memory leak will increase the size of the allocated, but unusable, memory.

Memory leaks can be difficult errors to detect but may be avoided by remembering to allocate and deallocate storage in the same procedure.

Assigning array values

```
a=0           !whole array set to zero
b(1)=5        !element 1 set to 5
c(j(2))=a(1)  !j(2) element of c set to a(1)
f(3,4)=a(10)  !2 d element (3,4)
```


What is this going to do?

```
integer, parameter :: nmax=100
real*8 array1d(nmax), array2d(nmax,nmax)

array1d(1)=1
array1d(nmax+1)=nmax+1
array2d(1,1)=1
array2d(0,0)=-1

print *, ' array1d ', array1d(1),
array1d(nmax+1)
print *, ' array2d ', array2d(1,1), &
array2d(0,0)=
end
```

I should have you write a small program to try it

(But that would take 20 minutes)

So here's the result

array1d	1.0000000000	101.00000000
array2d	1.0000000000	-1.0000000000

So it seems like it worked!

The problem is that it is not guaranteed to work!

How about I go 10^9 outside of bounds

```
579 $ a.out
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image                PC                Routine           Line            Source
a.out                 000000010ED9BCFB  Unknown          Unknown         Unknown
a.out                 000000010ED9BA3C  Unknown          Unknown         Unknown
libdyld.dylib        00007FFF8F3BE7E1  Unknown          Unknown         Unknown
```

Here we lucked out and the OS protected us from ourselves.

It looks like we wanted to write into the dynamic library (which is shared by all the programs running on the machine – probably not a nice thing to do).

This “problem” is not restricted to Fortran.

C, and most other languages, will do the same bad things very nicely also.

Produces the famous “segmentation fault”

(says you are trying to trespass outside your “property” in memory, and it will not let you).

But it does let you do it within your “property” (in the memory allocated to your program).

This may or may not cause your program to get bad results or crash.

How to fix it?

Use something called “bounds checking”

Most languages do not do it by default as it is incredibly slow.

However, you can tell the compiler (it takes extra code) to do it, and it will then save you from yourself.

```
583 $ ifort -check bounds dynamem.f90
```

```
584 $ a.out
```

```
forrtl: severe (408): fort: (2): Subscript #1 of the array  
ARRAY1D has value 101 which is greater than the upper  
bound of 100
```

```
...
```

Can do “scalar” operations on arrays (as in
Matlab: `.+`, `.-`, `.*`, `./`)

For arrays `a`, `b` and `c` (the arrays have to be
declared earlier)

`c=a+b`

Arrays have to be “conformable” – same size.

Subroutines and Functions (subprograms)

Functions return a single value

Subroutines can return multiple values through an argument list.

Functions – roll your own

```
program testfunctions
implicit none
real a, b, a_mean, g_mean
write(*,*) 'enter two real values'
read(*,*) a,b
a_mean=ArithMean(a,b)
g_mean=GeoMean(a,b)
write(*,*) 'value 1 ',a,' value 2 ',b,' arith mean ',a_mean,'
geom mean ',g_mean
```

```
contains
real function ArithMean(a,b)
implicit none
real, intent(in) :: a,b
ArithMean=(a+b)/2.0
end function ArithMean
```

```
real function GeoMean(a,b)
implicit none
real, intent(in) :: a,b
GeoMean=sqrt(a*b)
end function GeoMean
```

```
end program testfunctions
```

Can put functions inside
same file as main program

– use **contains**

Can also put in another
file and combine at
compile time.

Intrinsic Functions

Built into Fortran (don't need to link libraries for I/O, math, etc.)

There are about a hundred of them, some of the most common

`sqrt`
`sin`, `cos`, `tan` – take argument in radians
`sind`, `cosd`, `tand` – take argument in degrees

Etc. – look 'em up.

```
program testsubroutines
implicit none
real a, b, c, d
write(*,*) 'enter two real values'
read(*,*) a,b
call Means(a,b,c,d)
write(*,*) 'value 1 ',a,' value 2 ',b,' arith mean ',c,' geom
mean ',d
```

```
contains
subroutine Means(a,b,c,d)
implicit none
real, intent(in) :: a,b
real, intent(out) :: c,d
c=(a+b)/2.0
d=sqrt(a*b)
end subroutine Means

end program testsubroutines
```

Subroutines

Similar to function –
but output now through
argument list, and can
have multiple outputs.

In example, a and b are
input and c and d are
output variables.

What is

`intent (xxx)`

where `xxx` is one of `in`, `out`, `inout`

`INTENT(IN)` function takes the value from the corresponding “formal argument” (the thing in the argument list in the subroutine definition) and does not change its content.

Is optional

Also have

`INTENT(OUT)` the “formal argument” does not receive a value from the calling program, but will return a value to the calling program through the corresponding argument.

`INTENT(INOUT)` the “formal argument” can both receive and return a value through the corresponding argument.

Both are optional and all are mutually exclusive

But – can get in trouble if don't specify
(problem is when put constant or expression in call)

```
real a
write(*,*) 'enter real value'
read(*,*) a
call MySub(2*a)
write(*,*) 'value a ',a
call MySub(2)
write(*,*) 'value a ',a
```

Contains

```
subroutine MySub(x)
implicit none x
real x=sqrt(x)
end subroutine Means
end
```

Main & subprograms don't have to be in same file

File mainsubs.f90

```
program mainsubs
implicit none
real a, b, c, d
write(*,*) 'enter two &
real values'
read(*,*) a,b
call Means(a,b,c,d)
write(*,*) 'value 1 '&
,a,'value 2 ',b&
,' arith mean ',c&
,' geom mean ',d
end program mainsubs
```

File subsubs.f90

```
subroutine
Means(a,b,c,d)
implicit none
real, intent(in) :: a,b
real, intent(out) :: c,d
c=(a+b)/2.0
d=sqrt(a*b)
end subroutine Means
```

To compile – list all source files needed

```
gfortran mainsubs.f90 subsubs.f90 -o myprog
```

Fortran passes by “reference” (address).

When you pass a variable to a subroutine it gets the address of the variable.

(use the `intent` statement to control what can be changed)

When you pass an array, the subroutine gets the address of the start of the array. There is no metadata. The subroutine does not know anything about the size of the array, so you also have to pass the size with more arguments.

Can resize an array in subroutine (dangerous).

So if you change the value of a variable in your Fortran subroutine (and you are not using intent) – the change is seen outside

(that's how you pass stuff back out!)

C, on the other hand, passes by “value” – a copy, so changes are local to inside the subroutine – except for arrays, which C also passes by reference.

Write a subroutine to multiply two arrays.

Write a program to do Gaussian elimination.
(should also be a subroutine or function).