

# Fortran Vs C (matlab)

- Logic is same for both languages
- Easy to learn and takes only a couple of days.
- No special commands for formatting statements & not case sensitive .
- Free version is available.

G95 is a good (and free) Fortran90 compiler which is very portable. So it can be used on most common platforms like Windows, Mac and Linux. See g95

Another good option is ifort (Non-commercial use).

Go to Intel.com for Fortran and Mac OS

# Quick run in Linux platform

Open a new file in any plain text editor

Types of editor: vim, gedit, emacs, kwrite etc (Type in a Terminal)

example: (for vim editor)

```
$ vim my_program.f90/my_program.f (*.f90 is preferable)
```

```
:i to insert text in vim editor
```

NB. if \*.f then start from 7 column

```
program main
```

Otherwise no preferences

```
print *, "hello, world!"
```

```
end
```

```
: wq # save and exit
```

```
$ifort/f95/gfortran/g95 my_program.f
```

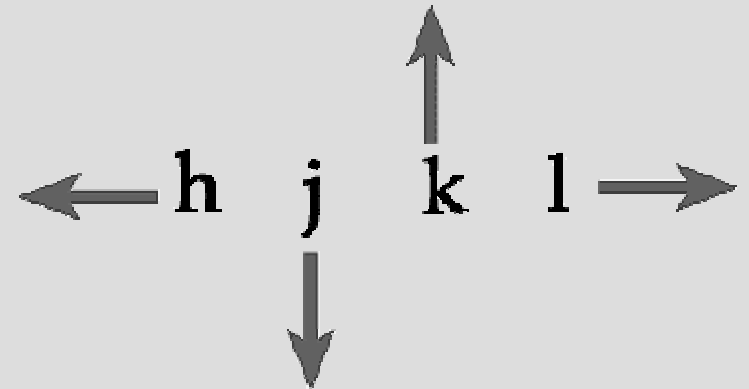
```
$hello, world!
```

# Useful commands in vim editor

## Exiting form vim

ZZ Write (if there were changes), then quit  
:wq Write, then quit  
:q Quit (will only work if file has not been changed)  
:q! Quit without saving changes to file

k Up one line  
j Down one line  
h Left one character  
l Right one character (or use <Spacebar>)  
w Right one word  
b Left one word  
i Enter text entry mode  
x Delete a character  
dd Delete a line  
r Replace a character  
R Overwrite text, press <Esc> to en



## Searching for word, sentence

/magic search for the word "magic" (case sensitive)  
n Repeat last search in the same direction  
:\$ Go to the end of the file  
:n Go to line with this number (:10 goes to line 10)  
:5,30d Delete lines 5 through 30

**Important**

# Continue ....

## COPY AND PASTE

<code>:5,5y</code>	Copy lines 5-10
<code>move cursor</code>	
<code>:put</code>	paste after cursor

## Substitution

<code>:%s/old/new/gc</code>	Substitutes old with new throughout the file
<code>:\$s/old/new/gc</code>	Substitutes old with new from the current cursor position to the end of the file
<code>:^,.s/old/new/gc</code>	Substitutes old with new from the beginning of the file to the current cursor position
<code>:&amp;</code>	Repeats the last substitute (:s) command

## NB:

`g` (global) is optional. It indicates you want to substitute all occurrences on the indicated lines. If you do not use `g`, the editor substitutes only the first occurrence on the indicated lines.

`c` (confirm) is optional. It indicates you want to confirm each substitution before `vi` completes it.

If vim is too difficult, use `kwrite` or `gedit` which have Windows-like features

# IMPLICIT NONE

- No assumptions about variable type (REAL or INTEGER).
- Forces all variables to be declared.
- Highly recommended for all programs.

example:

```
Program main
```

```
Implicit none
```

```
....
```

```
....
```

```
end
```

# Constants and Variable

The main types of variables we need in F90 are

integer, real

and complex, declared thus,

```
integer :: i;
```

```
real*8 :: x, y, z(1:20);
```

15 digits accuracy after decimal point

```
complex*16 :: cx, crhs(1:10);
```

For constants, use the 'parameter' keyword,

```
real*8, parameter :: pi = 3.141592654d0 ;
```

```
real*8,parameter :: AA =2.0d0, AAA=5.00d0 ;
```

```
integer, parameter :: N = 128, ... ;
```

```
parameter (x=1.0, y=2.0,cx=(1.0,2.0))
```

Dont forget to put d0 to  
make make sure that it  
actually real\*8

## Allocatable

```
REAL, ALLOCATABLE :: A(:)
```

```
READ(*,*) N
```

```
ALLOCATE (A(N))
```

```
...
```

```
DEALLOCATE (A)
```

# More about arrays

As we saw in the previous slide, arrays can be declared as follows:

```
real*8 u(1:100);
```

```
real*8 v(1:100, 1:100);
```

here 'v' is a 2D array of reals.

We can also have constant parameter arrays,

```
real*8, parameter :: a(1:3) = (/ 0, -5/9,  
-153/128 /);
```

```
real*8 x(1:10);
```

Arrays need not be 1-based, i.e. the first element need not be 1

```
real*8 y(-10:10);
```

will declare a 21-element array of reals spanning  $y(-10)$ ,  $y(-9)$ , ...  $y(0)$ , ...  $y(9)$ ,  $y(10)$

To initialize an array to the same value (for e.g. 0),

```
real*8 z(1:10);
```

```
z = 0;
```

# Character and Logical

```
CHARACTER (LEN=LL) :: sign !specify its length by an  
integer LL
```

```
CHARACTER(LEN=3), PARAMETER :: day(0:6) = &  
    (/'Sun','Mon','Tue','Wed','Thu','Fri','Sat'/)  
    ! array constant
```

```
logical a, b  
a = .TRUE.  
b = .FALSE.
```

# Loops in Fortran

## do ... end do

```
do i = 1,10
print *, "i = ", i, "sin &
(i)", sin(real(i));
end do
```

To perform reverse loops,

```
do j = 10,1,-1
print *, "j = ", j;
end do
```

or loops with steps not  
equal to 1,

```
do k = 2,100,2; print *, k;
end do
```

## do while ... end do

```
count = 0
DO WHILE (count <= max)
    x(count) = count
    count = count + 1
END DO !while_loop ended
```

NB:

EXIT ! comes out from the loop

CONTINUE/CYCLE ! keeps on running

# Continue ... looping

case

```
SELECT CASE (exp)

    CASE (0)
        X = 1.0

    CASE (1)
        X = 10.0

    CASE (2)
        X = 100.0

    CASE DEFAULT
        X = -1.0

END SELECT
```

# Control Constructs/ decision

<	.LT.	>	.GT.
<=	.LE.	>=	.GE.
==	.EQ.	/=	.NE.

**Conditional  
Expressions**

## If loop

```
IF (x .GT. 3) THEN
    CALL SUB1
ELSEIF (x .EQ. 3) THEN
    A = B*C-D
ELSEIF (x .EQ. 2) THEN
    A = B*B
ELSE
    A=B
ENDIF
```

# Pointer

# A variable that has the POINTER attribute can point to a variable that has the TARGET attribute.

# Linked by the new pointer-association operator '=>'

```
REAL, POINTER    :: ptr
REAL, TARGET     :: tgt
```

```
ptr => tgt
```

## example

```
PROGRAM POINTERS
```

```
REAL, POINTER    :: A, B
```

```
REAL, TARGET     :: t1 = 1.1, t2 = 2.2
```

```
    A => t1
```

```
! A -> t1
```

```
    B => t2
```

```
! B -> t2
```

```
    print * , A, B, t1, t2
```

```
END PROGRAM
```

```
1.10      2.20      1.10      2.20
A          B          t1          t2
```

# Some Useful commands

## EQUIVALENCE:

two or more variables can share same memory location

```
REAL*8                :: U1(1:200,1:200,1:200), CU1  
(1:200,1:200,1:200)
```

```
REAL*8                ::
```

EQUIVALENCE (U1, CU1,AFI)!u1,cu1,af1 are pointing same memory

SIZE(ARRAY) ! is a function which returns the number of  
! elements in an array ARRAY

SHAPE(ARRAY) ! is a function which returns shape of an  
array ARRAY

example:

```
real*8  ::A(1:4,1:6,1:8)
```

```
size(A) => 192
```

```
shape(A) => 4 6 8
```

# Subroutine

Subroutines help us break down complex programs into units of simpler tasks. For eg,

```
program main
real*8 :: x, y, z;
x = 1; y = 2;
call add_em(x, y, z);
print *, "z = ", z;
end

subroutine add_em(a, b, c)
real*8 :: a, b, c;
c = a + b;
end
```

# More . . . subroutine

Header file ! we can make our header file when large number of big  
! sized arrays are required to pass

Example: "header" # name of the file, place in same directory

```
IMPLICIT NONE
INTEGER NX, NY, NZ, N_TH
PARAMETER (NX=64,NY=256,NZ=64)
LOGICAL USE_MPI
REAL*8 NU, LX, LY, LZ
REAL*8 U1(1:NX,1:NY,1:NZ), U2(1:NX,1:NY,1:NZ),
        U3(1:NX,1:NY,1:NZ)
COMMON /INPUT/ NU, LX, LY, LZ, U1, U2, U3
```

```
Program main
include 'header'
write(*,*) NX
end
```

```
Subroutine
include header
write(*,*) NX
end
```

# More ... subroutine

```
Module global
IMPLICIT NONE
INTEGER NX, NY, NZ, N_TH
PARAMETER (NX=64,NY=256,NZ=64)
LOGICAL USE_MPI
REAL*8 NU, LX, LY, LZ
REAL*8 U1(1:NX,1:NY,1:NZ), U1(1:NX,1:NY,1:NZ),
        U3(1:NX,1:NY,1:NZ)
COMMON /INPUT/ NU, LX, LY, LZ, U1, U2, U3
contains
subroutine
...
end
end module global

Program main
use global
write(*,*) NX
end
```

# More . . . module

If module file is used by many programs, then make a separate file for module along with other subroutines

Main file: my\_program.f95

Module file:  
module\_file.f95

Compile module file along with main program

```
$ f95 my_program.f95 module_file.f95 -o my_program
```

```
$ ./my_program
```

# Input / OUTPUT statement

To open a new file to output data,

```
open (unit=10, file="output.txt", status='new');  
write (10,*) x;  
close(10);
```

To open a old file to read data,

```
open(40,file="output.txt",form='formatted',status='old')  
read (40,*) x; close(40)
```

To open a old file to add more data,

```
open(40,file="output.txt",form='formatted',  
status='old', position='append')
```

#POSITION="APPEND" - to append to an existing sequential file.

#POSITION="REWIND" - to ensure that existing file opened at  
beginning (more portable)

#STATUS="REPLACE" to overwrite any old file or create a new one.

#STATUS="new" -to create a new one if file with same name is  
absent and not overwrite the existing file.

#STATUS="unknown" same as "REPLACE" -

#STATUS="old" used for reading data from old file or append  
data

# Format commands

## FORMAT

Example:

```
WRITE(*,60) A, X, Y,Z
```

```
60 FORMAT ( I6.4 , 2F10.6 , E12.3 )
```

I6.4 (for integer) specifies a total of 6 characters including a sign and leading spaces, if necessary, but at least 4 digits

and 2F10.6 (for real) specifies a total of 10 characters with 6 digits following the decimal point for 2 variables

and E14.6 (for real) 14 positions of which :

6 are used for the decimals, 4 - for the exponent

1 - for the sign, 1 - for the starting zero

1 - for the decimal point, 1 - for a blank character exponent

# Continuation of line in fortran90

Line can be up to 132  
characters long.

Line continuation is the & (ampersand) on the leading line.

```
print *, 'This statement is continued and will have&  
          ten spaces after have.'
```

Continuation without extra spaces:

```
print *, 'This statement is continued and will have&  
& only one space after have.'
```

# Comments

Use the ! character to comment code.

```
! this is a comment
```

```
! as is this
```

```
integer, parameter :: N = 128; ! points
```

The Fortran 90 compiler ignores everything starting from the ! character to the end of that line

Sprinkle your code generously with comments as that helps you and others understand the code better.

And it also helps make the TA's life easier while grading ;-)

# Make file

Makefiles automate the compiling/linking process for big projects. The make program looks for a file called 'Makefile' in the current folder and decides which programs need compiling

What does a make file do ?? It is meant to replace the sequence of commands in the file which would, otherwise, have to be typed at the terminal

```
$f95 -c module_file.f90      ! create object file: module_file.o and  
                             ! & module file: global.mod used by  
                             ! main program
```

```
$f95 -c my_program.f90      ! object file: my_program.o
```

```
ifort -o exeprogram module_file.o my_program.o
```

# More . . . Makefile

Create a file in same directory named "Makefile"

```
# This is an commentary line in a makefile
```

```
# Start of the makefile
```

```
COMPILER = f95
```

```
USEROPTS = -O3
```

```
COMPOPTS = $(USEROPTS)
```

```
my_program: my_program.f90 module_file.o
```

```
    $(COMPILER) $(COMPOPTS) my_program.f90 -o my_program \
    module_file.o
```

```
module_file.o: module_file.f90
```

```
    $(COMPILER) $(COMPOPTS) -c module_file.f90
```

```
clean:
```

```
    rm module_file.o my_program
```

```
# End of the makefile
```

# Lets play with linux/Unix

Commands are case sensitive

Change directory:

```
$ cd /home/username
```

Go back to home directory

```
$ cd ~
```

To know where you are:

```
$ pwd          ! print name of current/working directory
```

To know what are here

```
$ ls          !lists the files in the current working folder
```

To know size files in mb/gb unit

```
$ ls -hl
```

```
$df          !view filesystem disk space usage
```

# More . . .

Make directory:

```
$ mkdir folder
```

Remove file/directory:

```
$ rm file !remove file
```

```
$ rmdir/(rm -r) folder !remove directory
```

```
$rm -v file !same as before shows which file has been deleted
```

To make permission for a file to read, write and executable:

```
chmod category+permissions filename
```

```
chmod a+x myfile ! a for all user and x executable
```

```
chmod o+r !this grants other read permission to the file data. The  
command
```

```
xhmod +rwx ! read, write and executable permission to all
```

If + is replaced by a -, then those permissions are taken away from the categories.

# More . . . .

Creating a tar file: (same as zip)

tar option(s) archive\_name file\_name(s)

option(s) -cf ! -c stands for to create an archive

! -f follow the next argument i.e. Input files

-cvf ! additional -v to show on terminal what are the files  
! are in tar file

-cvjf ! more compression as form of file\_name.tar.bz2  
! similar as -z (for gzip)

example

```
$ tar -cvf file.tar myfile.txt
```

```
$ tar -cvjf file.tar.bz2 file1 file2
```

to unpack a tar file: (same unzip)

no untar command plz.. !!

-xvf ! add -x instead of -c to extract it from tar

example :

```
tar -xvf file.tar
```

```
tar -xvjf file.tar.bz2
```

Sequence of the options doesnt matter:

-cvjf = -vjcf (but f is always final one)