Introduction to Fortran

Adapted by Susan Allen from notes by:
Doug Sondak
SCV
sondak@bu.edu

Outline

- Goals
- Introduction
- Fortran History
- Basic syntax
- Additional syntax

Goals

- To be able to understand and modify existing Fortran code

Introduction

- Python is great! Why do I need to learn a new language?!
- All codes must be translated to machine language
- Interpreted language
  - Matlab, Python, Java
  - Translation is performed incrementally at run time
- Compiled language
  - Fortran, C, C++
  - Translation is performed once, then executable is run

Introduction (cont'd)

- Compiled languages run faster
  - D. Sondak translated a program from Matlab to C for a user, and it ran 7 times as fast
  - Large-scale computing is usually done with compiled language
- Some convenient features of interpreted languages (e.g., no need to declare variables) result in performance and/or memory penalties

Fortran History

- Before Fortran, programs were written in assembly language
- Fortran was the first widely-used high-level computer language
  - 1957
  - Developed by IBM for scientific applications

Fortran History

- WATFOR (1965)
- Fortran 66 (1966)
- WATFIV (1968)
- Fortran 77 (1978)
- Fortran 90 (1991)
  - “fairly” modern (structures, etc.)
  - Current “workhorse” Fortran
- Fortran 95 (minor tweaks to Fortran 90)
- Fortran 2003
  - Gradually being implemented by compiler companies
  - Object-oriented support
  - Interoperability with C is in the standard
  - yay!

What Language Should I Use?

- I *usually* suggest using the language you know best
- Python is great, but is not a good choice for major number crunching
  - Researchers often write codes in say Matlab, and they grow and grow (the codes, not the researchers) until they are much too slow
  - Then a painful translation is often required

What Language? (cont’d)

- Fortran is hard to beat for performance
- C has the potential to be as fast as Fortran if you avoid aliasing issues and promise the optimizer your code won’t screw up aliasing
  - Fortran doesn’t have this issue due to the different nature of its pointers
- I have not written large C++ codes, but it’s to my understanding that object-oriented constructs tend to be slow
- Suggestion – write computationally-intensive codes in Fortran or C
  - Can parallelize using MPI and/or OpenMP

Fortran 90+ Syntax

- Source lines are not ended with semicolons (as in C or Matlab)
- Ampersand at end of line tells compiler that statement is continued on next source line
- ***Not*** case-sensitive
- Spaces don’t matter except within literal character strings
  - I use them liberally to make code easy to read
- Comment character is **!**

Fortran 77 Syntax

- Columns matter. Continuation is line 6, line numbers in 2-5
- Source lines are not ended with semicolons (as in C or Matlab)
- ***Not*** case-sensitive
- Comment character is **c** in column 1

Examples

- gork.f : a fortran 77 file
- thermal_wind.f90 : a fortran 90 file
- Differences in layout

Fortran 90+ Syntax (cont'd)

- Declarations – should declare each variable as being integer, real (floating point), character, etc.
- Integers and reals are stored differently
- Integer arithmetic will truncate result
  - If i=3 and k=2, i/k=1, k/i=0

Fortran Syntax (3)

- There are sometimes several ways to do the same thing
  - For backward compatibility
  - We will only use modern constructs
- Source file suffix is compiler dependent
  - Usually .f90/.F90 for Fortran 90+, .f for Fortran 77

Fortran Syntax (4)

- First statement in code is program statement
  - Followed by program name
  - I like to give the source file the same name as the program
  - myprog.f90 (name of source file)
  - program myprog (first line in source code)

Fortran Syntax (5)

- implicit none
  - Due to older versions of Fortran
  - Had “implicit typing”
    - Variables did *not* have to be declared
    - If names started with i-n, were automatically integers
    - If a-h,o-z, were automatically single-precision reals
  - Implicit typing is considered bad programming practice
  - **Always use implicit none**
  - often next line after program
  - Implicit none says that all variables must be declared
  - If you use implicit none and don’t declare all variables, compiler will yell at you

Fortran Syntax (6)

- A character string is enclosed by single quotes
  - Characters within the quotes will be taken literally

‘This is my character string.’

- print*
  - “list-directed” output
  - Lazy way to produce output
  - No format required
  - Follow by comma, then stuff to print

```
print*, ’This is my character string.’
```

Fortran Syntax (7)

- At the end of the code is an end program statement, followed by program name
  - Paired with program statement that starts the code

```
end program myprog
```

Examples

- thermal_wind.f90
- Program statement
- Implicit none
- Character string “” instead of ' '
- write (*,*) instead of print*
- End program statement

Compilation

- A compiler is a program that reads source code and converts it to a form usable by the computer
- Code compiled for a given processor will not generally run on other processors
  - AMD and Intel *are* compatible
- On waterhole machines we have gnu fortran gfortran
- On westgrid we are using intel fortran mpif90
- On windows machines... hard to get
- Commerical fortran, I've used portland group

Compilation (cont’d)

- Compilers have huge numbers of options
- For now, we will simply use the –o option, which allows you to specify the name of the resulting executable
- On the command line:

```
gfortran  –o  thermal_wind.exe  thermal_wind.f90
```

Compilation (4)

- Compile your code
- If it simply returns a command line prompt it worked
- If you get error messages, read them carefully and see if you can fix the source code and re-compile
- Once it compiles correctly, type the executable name at a command line prompt: **./thermal_wind.exe**

Example

- Compile and run thermal_wind.f90

Declarations

- Lists of variables with their associated types
- Placed in source code directly after “implicit none”
- Basic types:
  - Integer
  - Real
  - Character
  - logical

Declarations (cont'd)

- Examples:

```
integer :: i, j, k
real :: xval, time
character(20) :: name, date
logical :: isit
```

Arithmetic

- +, -, *, /
- ** indicates power

  $2.4^{1.5} \longrightarrow$ 2.4**1.5

- Built-in functions such as sin, acos, exp, etc.
- Exponential notation indicated by letter “e”

  $4.2 \times 10^{3} \longrightarrow$ 4.2e5

More List-Directed i/o

- read* is list-directed read, analogous to print*
- Follow with comma, then comma-delimited list of variables you want to read

```
read*, x, j
```

- Often use list-directed read and write together

```
print*, 'Enter a float and an integer:'
read*, x, j
print*, 'float = ',  x, '   integer = ', j
```

Examples 2

- thermal_wind.f90
- Declarations
- Arithmetic
- Built in functions
- Read statemet

Arrays

- Specify static dimensions in declaration:

```
real, dimension(10,3,5) :: x
```

- Starts at 1 like Matlab, not 0 like Python, C

- Can also specify ranges of values

```
integer, dimension(3:11, -15:-2) :: ival, jval
```

Arrays (cont'd)

- Dynamic allocation
  - Need to specify no. dimensions in declaration
  - Need to specify that it's an allocatable array

```
real, dimension(:,:,:), allocatable :: x, y
```

  - allocate function performs allocation

```
allocate(  x(ni,nj,nk),  y(ldim,mdim,ndim)  )
```

  - When you're done with the variables, deallocate

```
deallocate(x, y)
```

not necessary at very end of code; Fortran will clean them up for you

Parameters

- If variable has known, fixed value, declare as parameter and initialize in declaration

```
integer, parameter :: idim = 100, jdim = 200
```

  - *Compiler* substitutes values wherever variables appear in code
  - Efficient, since there are no memory accesses
  - Often used for declaring arrays

```
integer, parameter :: idim = 100, jdim = 200
real, dimension(idim, jdim) :: x
integer, dimension(idim) :: iarray
```

Examples 3

- gork.f
- Arrays (declaration)
- upwelling.f90
- Dynamic allocation

Control

- Do loop repeats calculation over range of indices

```
do i = 1, 10
     a(i) = sqrt( b(i)**2  +  c(i)**2 )
enddo
```

- Can use increment that is not equal to 1
  - Goes at *end* of do statement, unlike Matlab

```
do i = 10, -10, -2
```

If-Then-Else

- Conditional execution of block of source code
- Based on relational operators

< less than

> greater than

== equal to

<= less than or equal to

>= greater than or equal to

/= not equal to

.and.

.or.

If-Then-Else (cont'd)

```
if( x > 0.0  .and.  y > 0.0 ) then
    z = 1.0/(x+y)
elseif ( x < 0.0  .and.  y < 0.0) then
    z = -2.0/(x+y)
else
    print*, 'Error condition'
endif
```

Examples 5

- upwelling.f90
- Loops
- If else endif

Array Syntax

- Fortran will perform operations on entire arrays
  - Like Matlab, python, unlike C
- To add two arrays, simply use

```
c = a + b
```

- Can also specify array sections

```
c(-5:10) = a(0:15) + b(0:30:2)
```

  - Here we use b(0), b(2), b(4), ...

Array Syntax (cont'd)

- There are intrinsic functions to perform some operations on entire arrays
  - sum

sum(x) is the same as x(1) + x(2) + x(3) + …

  - product
  - minval
  - maxval

Subprograms

- Subroutines and functions
  - Function returns a single object (number, array, etc.), and usually does not alter the arguments
  - Altering arguments in a function, called “side effects,” is sometimes considered bad programming practice
  - Subroutine transfers calculated values (if any) through arguments

Functions

- Definition starts with a return type
- End with “end function” analogous to “end program”
- Example: distance between two vectors

```
real function distfunc(a, b)
    real, dimension(3) :: a, b
    distfunc = sqrt( sum((b-a)**2) )
end function distfunc
```

- Use:

```
z = distfunc(x, y)
```

  - Names of dummy arguments don’t have to match actual names
  - distfunc must be declared in calling routine

```
real :: distfunc
```

Subroutines

- End with “end subroutine” analogous to “end program”
- Distance subroutine

```
subroutine distsub(a, b, dist)
    real :: dist
    real, dimension(3) :: a, b
    dist = sqrt( sum((b-a)**2) )
end subroutine distfunc
```

- Use:

```
call distsub (x, y, d)
```

  - As with function, names of dummy arguments don’t have to match actual names

Examples 6

- trc_nam_my_trc.F90
- subroutines

Basics of Code Management

- Large codes usually consist of multiple files
- I usually create a separate file for each subprogram
  - Easier to edit
  - Can recompile one subprogram at a time
- Files can be compiled, but not linked, using –c option; then object files can be linked

```
gfortran –c mycode.f90
gfortran –c myfunc.f90
gfortran –o mycode mycode.o myfunc.o
```

Makefiles

- Make is a Unix utility to help manage codes
- When you make changes to files, it will
  - Automatically deduce which files need to be compiled and compile them
  - Link latest object files
- *Makefile* is a file that tells the make utility what to do
- Default name of file is “makefile” or “Makefile”
  - Can use other names if you’d like

Kind

- Declarations of variables can be modified using “kind” parameter
- Often used for precision of reals
- Intrinsic function selected_real_kind(n) returns kind that will have at least *n* significant digits
  - n = 6 will give you “single precision”
  - n = 12 will give you “double precision”

Kind (cont'd)

```
integer, parameter :: rk =
selected_real_kind(12)

real(rk) :: x, y, z

real(rk), dimension(101,101,101) :: a
```

- If you want to change precision, can easily be done by changing one line of code

Modules

- Program units that group variables and subprograms
- Good for global variables
- Checking of subprogram arguments
  - If type or number is wrong, linker will yell at you
- Can be convenient to package variables and/or subprograms of a given type

Modules (cont'd)

module *module-name*

implicit none

... variable declarations ...

contains

... subprogram definitions ...

end module *module-name*

Modules (3)

- Only need “contains” if module contains subprograms
- Doug Sondak usually names his modules (and associated files) with _mod in the name, e.g., solvers_mod, solvers_mod.f90
- In program unit that needs to access components of module use *module-name*
- use statement must be *before* implicit none

Modules (4)

- use statement may specify specific components to access by using “only” qualifier:

```
use solvers_mod, only: nvals, x
```

- A Fortran style suggestion:
  - Group global variables in modules based on function
  - Employ “use only” for all variables required in program unit
  - All variables then appear at top of program unit in declarations or “use” statements

Examples 7

- upwelling.f90
- Kind
- Module
- Use only
- public

Derived Types

- Analogous to structures in C
- Can package a number of variables under one name

```
type grid
    integer :: nvals
    real, dimension(100,100) :: x, y, jacobian
end type grid
```

Derived Types (cont'd)

- To declare a variable

```
type(grid) :: grid1
```

- Components are accessed using %

```
grid1%nvals = 20
grid1%x = 0.0
```

- Handy way to transfer lots of data to a subprogram

```
call  calc_jacobian(grid1)
```

i/o

- List-directed output gives little control
- write statement allows formatted output

write(unit, format) variables

- Unit is a number indicating where you want to write data
  - The number 6 is std out (write to screen)

i/o (2)

- Example write/format statement

```
write(6, 98) ‘answers are ’, x, j, y
98 format (a, f6.2, i5, es15.3)
```

i/o (3)

- Suppose you want to write to a file?
  - open statement

open(11, file=‘mydata.d’)

  - “11” is unit number
  - Don’t use 5 or 6
    - Reserved for std in, std out
  - Use this unit in your write statement
  - When you’re finished writing, close the file

close(11)

i/o (4)

- Can also read from file
  - read rather than write
  - Can use * instead of format specifier

```
read(11,*) j, x
```

Examples 9

- thermal_wind.f90
- open
- read
- write
- close
- trc_nam_my_trc.F90
- rewind
-

Unformatted i/o

- Binary data take much less disk space than ascii (formatted) data
- Data can be written in binary representation
  - Not directly human-readable

```
open(199, file='unf.d', form='unformatted')
write(199)  x(1:100000),  j1, j2
read(199)   x(1:100000),  j1, j2
```

  - Note that there is no format specification
- Fortran unformatted slightly different format than C binary
  - Fortran unformatted contains record delimiters

Citation

This set of slides was adapted from:

www.bu.edu/tech/files/2010/10/fortran.pptx

Preprocessors

Large codes often use C or fortran pre-processors to include/exclude code :: ie to customize the same code base for multiple applications

- The preprocessor statements start with a #
- You can define variables:
  - #define SOLARIS_2   .TRUE.
- Or undefine
  - #undef SOLARIS_2
  -

Preprocessors

Most often used it conditional source code selection:

```
#define SOLARIS_2        .TRUE.
#if (SOLARIS_2)
    CALL solaris_2 (X,Y,Z)
#else
    CALL solaris_1 (X,Y,Z)
#endif
```

Preprocessors

My source :
http://globalchange.bnu.edu.cn/upfile/fpp.pdf

Fortran in Python

My source :
http://nbviewer.ipython.org/github/mgaitan/fortran_magic/blob/master/documentation.ipynb