Topics

- Why Program in Fortran
- General Programming
- Variables/Datatypes
- Flow Control
- Programs
- Functions/Subroutines
- Modules
- Libraries
- Case Studies
Why Program?

- Many problems in engineering have no analytical solution
- Numerical methods can be used to solve many problems
- Modern personal computers have enough power for many engineering problems
Why Fortran?

- General programing language
- Designed for engineering
- Very good array support
- Easy memory management
- Readable syntax
- High performance
Programing vs Applications

- Programing allows for the creation of new applications
- Custom programs are more flexible than prebuild applications
- Special-purpose codes are generally faster than generic ones
- Applications can be easier sometimes
- Why reinvent the wheel?
General Programming

```c
typedef void (*AnjutaPluginValueRemoved) (AnjutaPlugin *plugin,
                                           const char *name,
                                           gpointer user_data);

#define ANJUTA_TYPE_PLUGIN (anjuta plugin get type (i))
#define ANJUTA_PLUGIN(o) ((G_TYPE_CHECK_INSTANCE_CAST ((o), ANJUTA_TYPE_PLUGIN, AnjutaPlugin*)) o)
#define ANJUTA_PLUGIN_CLASS(K) (G_TYPE_CHECK_CLASS_CAST (K, ANJUTA_TYPE_PLUGIN, AnjutaPluginClass))
#define ANJUTA.PluginPrivate(o) (G_TYPE_CHECK_INSTANCE_CAST ((o), ANJUTA_PLUGIN_CLASS, AnjutaPluginPrivate))
#define ANJUTA_PLUGIN_GET_CLASS(o) (G_TYPE_INSTANCE_GET_CLASS ((o), ANJUTA_TYPE_PLUGIN, AnjutaPluginClass))

struct AnjutaPlugin {
  GObject parent;

  /* The shell in which the plugin has been added */
  AnjutaShell *shell;

  /* private */
  AnjutaPluginPrivate *priv;
};
```

```shell
Compiling text_editor_prefs.c -> text_editor_prefs.o
Completed... successful
```
How does it work?

Source Code → Compiler → Object Code → Linker + Libraries → Executable Program
Source Code

- Human readable
- Just a text file
- File name extensions
  - Fortran 90/95: .f90 or .f95
  - Fortran 77: .f .for or .F
Compiler

- Specialized program
- Converts source code to object code
- Generally accept many parameters
  - Debugging symbols on/off
  - Aggressive/Passive optimization
  - Code generation options
Object Code

- Instructions the computer understands
- Unresolved references between objects
- Extensions
  - Windows: .obj
  - Unix: .o
Libraries

- Contain routines written previously
- Often needed for language features
  - Intrinsic functions are often in a library
- Allow for extensions to the standard
  - MPI
  - LAPACK/BLAS
  - FFTW
Linker

- Combines object code with libraries to make an executable file
- Resolves references between objects
Executable Program

- Just another file on the hard disk
- Specially formatted so the OS knows how to execute it
- Applications are just big programs
Variables/Datatypes
Variables

- Store a single value for later use
- Must be declared at beginning of code
- Store only one type of data

Examples:

INTEGER::i,j,k
REAL::x,y
CHARACTER(len=50)::name
LOGICAL::loop, isDone
Data Representation

- Computers store data in binary
- Numbers are integers or floating point
- Character type uses ASCII to store data
  - Table that converts letters to 8-bit binary
- Logical stores True/False values
Numbers

- Integers
  - Whole numbers only
  - Exact
  - KIND attribute specifies range

- Reals
  - Fractional numbers
  - Only approximations of actual value
  - KIND attribute specifies accuracy
Arithmetic

- Most mathematical functions are present
- Remember that x squared is x**2
- Many intrinsic functions (sin, cos, log, etc)
- Don't mix integers and reals
  - Use REAL(), FLOOR(), and CEILING() to convert between reals and integers
  - [real]**[integer] is ok
Example

INTEGER::k
REAL::x

k = 1+2
k = 5*k/2 !Note k==7

x = 1.0+2.0
x = 5.0*x/2.0 !Note x==7.5
Arrays

- Store many scalar values together as one
- Used Extensively in scientific computing
  - Can be used for field variables
    - Velocity fields in CFD
    - Temperature in heat transfer
  - Node displacements in FEM
  - Data tables
  - Many more...
Arrays in Memory

- Arrays are stored in blocks of memory
- Arrays are stored in column major order

Example:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix} \rightarrow [1\ 3\ 2\ 4]
\]
Array Notation

- Declare arrays
  
  ```
  REAL,DIMENSION(10)::x,y
  REAL,DIMENSION(3,3)::tensor
  ```

- Index Arrays
  
  ```
  x(5) = 1.0
  y(3) = x(5)
  tensor(2,3) = 3.0
  ```
Array Section Notation

- Entire sections of arrays can be used
  \[ x(1:3) = \text{tensor}(1,1:3) \]
  \[ \text{tensor}(1:2,2:3) = \text{tensor}(2:3,1:2) \]
  \[ x(:) = y(:) \]
Allocatable Arrays

- Sometimes array sizes depend on input data at runtime
- Allocatable statement deferrs array sizing from compile-time to runtime

```fortran
REAL, ALLOCATABLE, DIMENSION(:, :) :: temp
INTEGER :: rows, cols
READ(*, *) rows, cols
ALLOCATE(temp(rows, cols))
```
User Defined Types

- Programmer can create composite types out of pre-existing types
- Can't contain allocatable arrays
- Access members using the '%' operator
Example

TYPE::particle
    REAL,DIMENSION(2)::X
    REAL,DIMENSION(2)::P
    REAL::M
    REAL::Q
END TYPE

TYPE(particle)::e1

e1 = particle((/0.0,0.0/),(/0.0,0.0/),1.0,1.0)
e1%P = (/1.0,0.0,0.0,0.0/)
Flow Control
**IF THEN**

- Used to conditionally execute code
- Requires a logical input

IF (k==2) THEN !Note == is not =

!do some stuff here

END IF
IF THEN ELSE

- Adds a clause to be run if the test is false

IF (k==2) THEN
  !k==2 stuff
ELSE IF (k==3) THEN
  !k==3 stuff
ELSE
  !other stuff
END IF
**SELECT CASE**

- Alternative to long If Then Else If ... End
- Switch executed code based on an integer or character value

```plaintext
SELECT CASE(k)
    CASE(1)
        !k==1 stuff
    CASE(2:5)
        !k==2...5 stuff
END SELECT
```
Loops

- Allow code to be run multiple times
- Fundamental to scientific computing
**Do Loop with Index**

- Execute once for each value in a set of integers

!Print 1,3,5,7,9,11 on the screen
DO k=1,11,2
   WRITE(*,*), k
END DO
**DO WHILE**

- Execute while a condition is met

```plaintext
k=1
DO WHILE(k<10)
  IF (getNext()==1) k=k+1
END DO
```
Infinite Do Loop

- Loops forever
- Used as a main control loop for interactive programs
- Programmer must tell program to terminate or exit the loop at some point

DO
  !do this forever
END DO
EXIT and CYCLE

- Exit will exit from the current loop and continue execution thereafter.
- Cycle prematurely terminates this time through a loop, but continues the same loop afterwards.
Examples

DO
  WRITE(*,*) “Quit? (Y/N)”
  READ(*,*) response
  IF (response=='Y') EXIT
END DO
Examples

!Double all elements in a lower than 11
a = (/1, 5, 9, 13, 17/)
DO k=LBOUND(a,1),UBOUND(a,1)
   IF (a(k)>10) CYCLE
   a(k) = 2*a(k)
END DO
Named Loops

- Loops can be named to prevent ambiguity
- EXIT and CYCLE can target a specific loop

ioloop: DO
  READ(*,*) response
  IF (response=='Y') CYCLE iolooop
END DO iolooop
Programs
The Program Block

- Each program must only have one program block
- Execution begins with the program block
- From the program block, functions, subroutines, modules and libraries can all be accessed
Example

PROGRAM hello

    CHARACTER(3)::msg
    msg = “Hi!”
    WRITE(*,*) msg

END PROGRAM hello
Functions/Subroutines
Functions

- Receive input and return a single output
- Contain their own code and variable declarations
- Functions variables lose their value between uses of the function, and cannot be used outside the function
- Much like functions in mathematics
REAL FUNCTION f(x)
   INTEGER,INTENT(IN)::x
   f = x**2-1.0
END FUNCTION f

- Data returned by function is REAL
- Note the INTENT flag on the dummy variable x
- Set the return value by assigning to the function's name
Subroutines

- Much like functions, but don't return any values
- Commonly used to alter the input variables to provide results
Example

SUBROUTINE increment(k)
    INTEGER, INTENT(INOUT)::k
    k = k + 1
END SUBROUTINE increment

• Notice how the subroutine can change the input variable
• Again notice the INTENT flag in the dummy variable
Calling Functions and Subroutines

!use previously defined function and subroutine

PROGRAM tester
   REAL::x=1.0
   INTEGER::k=3
   x = f(2.0)   !x is now 3.0
   CALL increment(k)  !k is now 4
END PROGRAM tester
Modules
Why use Modules?

• Compartmentalize code
  - Keep like subroutines and functions together
  - Easier to understand
  - Interface checking

• Reuse code for multiple projects
Module Structure

- Multiple modules can exist in same file
- Modules can use other modules
- Contain type definitions, parameters, global variables, interfaces, functions and subroutines
Example

MODULE polar
  IMPLICIT NONE
  REAL,PARAMETER::PI=4.0*ATAN(1.0)
CONTAINS
  FUNCTION radius(x,y) RESULT(o)
    REAL,INTENT(IN)::x,y
    REAL::o
    o = SQRT(x**2+y**2)
  END FUNCTION radius
END MODULE polar
More on CONTAINS

- Programs may also have a contains section
- If a subroutine or function exists outside a program or module, it too may have a contains section
Libraries

• Many libraries are available for scientific computation
  – LAPACK
  – FFTW3
  – PGPLOT
  – FMLIB
  – OpenMPI
  – Many many more...

• Don't reinvent the wheel
Case Studies

- Newton's Method Root Finder
- Runga-Kutta Integration
- Heat in a Plate (Jacobi)
- Gaussian Elimination
Newton's Method
MODULE functions

IMPLICIT NONE

CONTAINS

REAL FUNCTION y(x)
     REAL,INTENT(IN)::x
     y = x**2-1.0
END FUNCTION y

REAL FUNCTION dydx(x)
     REAL,INTENT(IN)::x
     dydx = 2.0*x
END FUNCTION dydx

END MODULE functions
MODULE solver
  USE functions
  IMPLICIT NONE
  REAL,PARAMETER::eps=EPSILON(1.0)
CONTAINS
  REAL FUNCTION next(x,d)
    REAL,INTENT(IN)::x
    LOGICAL,INTENT(OUT)::d
    next = x-y(x)/dydx(x)
    d = .FALSE.
    IF(ABS(next-x)<x*eps) d=.TRUE.
  END FUNCTION next
END MODULE solver
Newton's Method (cont)

PROGRAM newton
    USE solver
    IMPLICIT NONE
    REAL::x = 10.0
    LOGICAL::done = .FALSE.
    DO WHILE(.not.done)
        x = next(x,done)
    END DO
    WRITE(*,*) x
END PROGRAM newton
Runga-Kutta Integration
R-K Integration (cont)

MODULE function
  IMPLICIT NONE
CONTAINS
  REAL FUNCTION dydt(y,t)
    REAL,INTENT(IN)::y,t
    dydt = SIN(t)
  END FUNCTION dydt
END MODULE function
MODULE rk_int

USE function
IMPLICIT NONE
CONTAINS

SUBROUTINE rk_step(y0,t0,h)

REAL,INTENT(INOUT)::y0,t0
REAL,INTENT(IN)::h
REAL::k1,k2,yn
k1 = dydt(y0,t0)
y0 = y0+k1*h
yn = y0+k1*h
k2 = dydt(yn,t0+h)
y0 = y0+h/2.0*(k1+k2)
t0 = t0+h

END SUBROUTINE rk_step

END MODULE rk_int
R-K Integration (cont)

PROGRAM runga_kutta

USE rk_int
IMPLICIT NONE
REAL,PARAMETER::h = 0.01
REAL,PARAMETER::PI = 3.14159265
REAL::y = 0.0, t = 0.0
DO WHILE(t<=2.0*PI)
   CALL rk_step(y,t,h)
   WRITE(*,*) t,y
END DO
WRITE(*,*) t,y
END PROGRAM runga_kutta
Heat in a Plate (Jacobi)
MODULE solver

IMPLICIT NONE
CONTAINS

SUBROUTINE step(T,err)
REAL,DIMENSION(:,:),INTENT(INOUT)::T
REAL,INTENT(OUT)::err
REAL::Tn
INTEGER::r,c
err = 0.0
DO c=LBOUND(T,2)+1, UBOUND(T,2)-1
   DO r=LBOUND(T,1)+1, UBOUND(T,1)-1
      Tn = 0.25*(T(r+1,c)+T(r-1,c)+T(r,c+1)+T(r,c-1))
      err = MAX(ABS(Tn-T(r,c)),err)
      T(r,c) = Tn
   END DO
END DO
END SUBROUTINE step
END MODULE solver
Jacobi (cont)

MODULE temperature
  IMPLICIT NONE
  REAL,DIMENSION(100,100)::T
CONTAINS
  SUBROUTINE initialize
    T(:,,:) = 50.0
    T(1,:) = 100.0
    T(100,:) = 25.0
    T(:,1) = 75.0
    T(:,100) = 0.0
  END SUBROUTINE initialize
END MODULE temperature
Jacobi (cont)

PROGRAM jacobi
    USE solver
    USE temperature
    IMPLICIT NONE
    REAL::err = 1.0
    CALL initialize
    DO WHILE(err > 1.0E-4)
        CALL step(T,err)
        WRITE(*,*) err
    END DO
END PROGRAM jacobi
Gaussian Elimination
Gaussian Elimination (cont)

REAL,DIMENSION(::,:),INTENT(IN)::aa
REAL,DIMENSION(SIZE(aa,1)),INTENT(IN)::bb
REAL,DIMENSION(SIZE(bb)),INTENT(OUT)::x
REAL,DIMENSION(SIZE(aa,1),SIZE(aa,2))::A
REAL,DIMENSION(SIZE(bb))::B
REAL,DIMENSION(SIZE(aa,2))::row
REAL::tmp, large, small, r
INTEGER::j,k
Gaussian Elimination (cont)

A = aa
B(:, :) = bb(:, :)
large = 0D0
DO k=LBOUND(A,1), UBOUND(A,1)
   large = MAX(large, ABS(A(k,k)))
END DO
small = EPSILON(1D0)*large*1D1
Gaussian Elimination (cont)

DO k=LBOUND(A,1),UBOUND(A,1)-1,1
    IF(ABS(A(k,k))<=small) THEN
        j = k+1
        DO WHILE(ABS(A(j,k))<=small)
            j = j+1
        END DO
        row(:) = A(k,:)
        A(k,:) = A(j,:)
        A(j,:) = row(:)
        tmp = B(k)
        B(k) = B(j)
        B(j) = tmp
    END IF
    B(k) = B(k)/A(k,k)
    A(k,:) = A(k,:)/A(k,k)
    DO j=k+1,UBOUND(A,1),1
        r = A(j,k)/A(k,k)
        A(j,:) = A(j,:)-r*A(k,:)
        B(j) = B(j)-r*B(k)
    END DO
END DO
Gaussian Elimination (cont)

\[
k = \text{UBOUND}(A,1)
\]
\[
B(k) = B(k)/A(k,k)
\]
\[
A(k,k) = 1D0
\]
\[
\text{DO} \ k = \text{UBOUND}(A,1), \text{LBOUND}(A,1)+1,-1
\]
\[
\quad \text{DO} \ j = k-1, \text{LBOUND}(A,1), -1
\]
\[
\quad \quad B(j) = B(j) - A(j,k)*B(k)
\]
\[
\quad \quad A(j,k) = 0D0
\]
\[
\quad \text{END DO}
\]
\[
\text{END DO}
\]
\[
\text{large} = \text{MAXVAL}(\text{ABS}(B))
\]
\[
\text{small} = \text{EPSILON}(1D0) * \text{large} * 1D1
\]
\[
\text{WHERE} (\text{ABS}(B) \leq \text{small})
\]
\[
\quad B = 0D0
\]
\[
\text{END WHERE}
\]
\[
x = B
\]