The Fortran 90 programming language

- Fortran has evolved since the early days of computing
- Fortran 90/95 is a modern programming language
- Many useful features for scientific (numerical) computing
- Widely used language in computational science
  - also widely used in finance, engineering, etc.
- Many “canned” subroutines available (often Fortran 77)
- Relatively easy to learn

Fortran 90 compilers

- Many commercial compilers available; often expensive
  - f90 available on buphy (Physics Dept. server)
- gfortran; free open source Fortran 90/95/2003/2008 compiler
  - part of gcc (gnu compiler collection)
  - installed on Physics Dept server buphy (and buphy0)
- g95; other open source product

Fortran 90 language tutorial

- Introduction to basic elements needed to get started

- Simple examples used to illustrate concepts

- Example programs also available on the web site

- For more complete language description, see, e.g.,
  - Fortran 90/95 explained, by M. Metcalf and J. Reid
  - Fortran 90/95 for Scientists and Engineers, by S. Chapman
  - Links on course web site (On-line Fortran resources)

Discussion and practice at Friday tutorials

To create a Fortran 90 program:

- Write program text to a file, using, e.g., Emacs

```
[program program-name]
   program statements
         …
end [program program-name]
```

- Compile & link using Fortran 90 compiler
  - creates "object" (.o) files
  - object files linked to form executable (.out or .x) file

Compilation/linking (using gfortran)

```
> gfortran program.f90
```

(gives executable program `a.out`)

```
> gfortran program.f90 -o program.x
```

(gives executable named `program.x`)

```
> gfortran -O program.f90
```

(turns on code optimization)

```
> gfortran -O program1.f90 program2.f90
```

(program written in more than one file)

```
> gfortran -O program1.f90 program2.o
```

(`unit` program2 previously compiled)

Variables and declarations

Intrinsic variable types

- integer
- real (floating-point)
- complex
- logical (boolean)
- character, character string ("text")
- arrays of all of these (up to 7-dimensional)

- ➢ A declaration is used to state the type of a variable
- ➢ Without declaration, `real` assumed, except for variables with names starting with i,…,n, which are `integer`
- ➢ Declarations forced with `implicit none` statement
  - always use this; eliminates 99% of programming errors!
- ➢ Fortran 90 is case-insensitive

Integers

A standard integer uses 4 bytes, holds numbers -2^{31} to $2^{31}-1$

```
integer :: i

i=-(2**30-1)*2-2
print*,I
i=i-1
print*,i
```

Output:

```
-2147483648
 2147483647
```

Note how -2^{31} has been written to stay within range!

i=(-2)**31 also works

i=-2**31 should not work

A “long” integer, `integer(8)`, is 8 bytes, holds -2^{63} to $2^{63}-1$

“Short” integers: `integer(2)`, `integer(1)`

Integer division: `3/2=1`, but `3./2=1.5`

We discuss the bit representation of numbers (on the board)

Floating-point numbers

A simple program which assigns values to real variables (single- and double precision; use 4 and 8 bytes):

```
implicit none

real    :: a
real(8) :: b

a=3.14159265358979328
print*,a
b=3.14159265358979328
print*,b
b=3.14159265358979328d0
print*,b

end
```

Output:

```
3.14159274
3.141592741012573
3.141592653589793
```

`3.14159265358979328_8`

is another way to specify double precision (8 bytes)

Complex numbers

Assignment of real and imaginary parts: `a=(ar,ai)`

```
complex :: a

a=(1.,2.)
print*,a,real(a),aimag(a)
a=a*(0.,1.)
print*,a
```

Output:

```
(1.,2.), 1., 2.
(-2.,1.)
```

`real(a)` and `aimag(a)` extract real and imaginary parts

Double precision: `complex(8)`

Characters and character strings

Example of characters, strings and operations with them:

```
integer         ::  n
character       ::  a,b
character(10)   ::  c

a='A';   b='B';   c=a//b//b//a;  n=len_trim(c)

print*,'Number of characters in c',n
print*,c(1:n),' ',c(2:3),' ',c(1:1)
print*,iachar(a),iachar(b)
print*,char(65),char(66),char(67),char(68)
```

Output:

```
Number of characters in c: 4
ABBA  BB  A
65, 66
ABCD
```

Logical (boolean) variables

Values denoted as `.true.` and `.false.` in programs
In input/output; values are given as `T` and `F`

Examples of boolean operators:
and, or, neqv (same as exclusive-or), not:

```
logical :: a,b

print*,'Give values (T/F) for a and b'
read*,a,b
print*,a.or.b,a.and.b,a.eqv.b,a.neqv.b,.not.a
```

Running this program ⇒

```
Give values (T/F) for a and b
T F
T F F T F
```

Arrays

Can have up to 7 dimensions. Example with integer arrays:

```
integer, dimension(2,2) :: a,b
integer :: c(2)

a(1,1)=1;  a(2,1)=2;  a(1,2)=3;  a(2,2)=4
b=2*a+1
print*,a
print*,b

c=a(1,1:2)
print*,c
```

Output:

1, 2, 3, 4
3, 5, 7, 9
1, 3

Lower bound declaration:

```
Integer :: a(-10:10)
```

Kind type parameter

For simplicity, a feature of type declarations was neglected

- "8" in `real(8)` does not actually refer to the number of bytes - it is a **kind type parameter**

With most compilers, the kind type parameter corresponds to the number of bytes used, but it does not have to

- with some compilers 1=single and 2=double precision

More generic way to declare a real:

- `real(selected_real_kind(m,n))`
  m = number of significant digits, n = exponent (10^{-n} - 10^{n})
  the type capable of representing at least this range and precision will be selected by the system (error if impossible)
- `real(kind(1.d0))`
  the function kind(a) extracts the kind type parameter of a

Analogous for integers: `selected_integer_kind(n)`

In this course we will for simplicity assume that the kind type parameter corresponds to the number of bytes (4,8 used)

Program control constructs

- Branching using `if … endif` and `select case`
- loops (repeated execution of code segments); `do … enddo`
- "Jumps" with `goto label#`

Branching with "`if … endif`"

```
If (logical_a) then
  statements_a
elseif (logical_b) then
  statements_b
…
else
  statements_else
endif
```

Relational operators

`==`	`.eq.`
`/=`	`.ne.`
`>`	`.gt.`
`<`	`.lt.`
`>=`	`.ge.`
`<=`	`.le.`

- Expressions `logical_i` take the values `.true.` or `.false.`
- Only statements after first true expression executed
- The `else` branch optional

Simpler form: `if (logical_expression) statement`

Example program; if.f90

```
integer :: int

print*,'Give an integer between 1 and 99'; read*,int
if (int<1.or.int>99) then
  print*,'Read the instructions more carefully! Good bye.'
elseif (int==8.or.int==88) then
  print*,'A lucky number; Congratulations!'
elseif (int==4.or.int==13) then
  print*,'Bad luck...not a good number; beware!'
else
  print*,'Nothing special with this number, '
  if (mod(int,2)==0) then
    print*,'but it is an even number'
  else
    print*,'but it is an odd number'
  endif
endif
```

Loops

Repeated execution of a code segment. Examples:

Standard loop (also valid in f77)

```
do i=1,n
  print*,i**2
enddo
```

"Infinite" loop

```
i=0
do
   i=i+1
   print*,i**2
   if (i==n) exit
enddo
```

Loop with `do while`

```
i=0
do while (i<n)
   i=i+1
   print*,i**2
enddo
```

"Jump" with `go to`

```
10 i=i+1
   i2=i**2
   if (i2<sqmax) then
     print*,i,i2
     goto 10
   endif
```

Procedures; subroutines and functions

- Program units that carry out specific tasks
- Fortran 90 has internal and external procedures

Internal subroutine

```
program someprogram
...
call asub(a1,a2,...)
...
contains
  subroutine asub(d1,d2,...)
  ...
  end subroutine asub
end program someprogram
```

- `asub` can access all variables of the main program
- `d1,d2` are "dummy" arguments

```
character(80) :: word

print*,'Give a word'; read*,word
 call reverse
 print*,word

 contains

   subroutine reverse

   implicit none

   integer :: i,n
   character(80) :: rword

   rword=''
   n=len_trim(word)
   do i=1,n
     rword(i:i)=word(n-i+1:n-i+1)
   end do
   word=rword

   end subroutine reverse

 end
```

Program writerev1.f90

- Subroutine call without an argument list
- The string `word` can be accessed directly since `reverse` is an internal subroutine

`len_trim(string)` gives length of `string` without trailing blanks

```fortran
character(80) :: word1,word2

 print*,'Give two words'; read*,word1,word2
 call reverse(word1)
 call reverse(word2)
 print*,trim(word2),' ',trim(word1)

contains

  subroutine reverse(word)

  implicit none

  integer :: i,n
  character(80) :: word,rword

  rword=''
  n=len_trim(word)
  do i=1,n
    rword(i:i)=word(n-i+1:n-i+1)
  enddo
  word=rword

  end subroutine reverse

 end
```

Program writerev2.f90

- Subroutine calls with argument lists
- Strings `word1,word2` are passed through the dummy variable `word`

`trim(string)`
string obtained when trailing blanks removed from `string`

```
character(80) :: word1,word2

print*,'Give two words'; read*,word1,word2
call reverse(word1(1:len_trim(word1)),len_trim(word1))
call reverse(word2(1:len_trim(word2)),len_trim(word2))
print*,trim(word2),' ',trim(word1)

end

subroutine reverse(word,n)

implicit none

integer :: i,n
character(n) :: word,rword

rword=''
do i=1,n
  rword(i:i)=word(n-i+1:n-i+1)
enddo
word=rword

end subroutine reverse
```

Program writerev3.f90

- External subroutine; cannot access variables of main program
- string `word` declared with variable length `n` passed from main

Functions (external)

```
function poly(n,a,x)

implicit none

integer :: i,n
real(8) :: poly,a(0:n),x

poly=0.0d0
do i=0,n
  poly=poly+a(i)*x**i
enddo

end function poly
```

main program:

```
...
integer :: n
real(8) :: a(0:nmax),x
real(8), external :: poly
...
print*,poly(n,a(0:n),x)
```

Accessing "global data"

Common blocks (outdated f77, but some times useful)

Global data accessible in any unit in which declarations and `common/blockname/v1,v2,...` appears

```
integer :: a,b
common/block_1/a,b
```

Modules

Global data accessible in any unit in which `use module_name` appears

```
module module_name
  integer :: a,b
end module module_name
```

Modules can also contain procedures, which are accessible only to program units using the module

Intrinsic procedures

- Many built-in functions (and some subroutines)
- In F90, many can take array argumens (not in F77)

Mathematical functions:

```
exp(x),sqrt(x),cos(x),...
```

Type conversion:

```
int(x),real(x),float(x)
```

Character and string functions:

`achar(i)` - ASCII character `i`

`iachar(c)` - # in ASCII sequence of character `c`

```
len(string),len_trim(string),trim(string)
```

Matrix and vector functions:

```
sum(a), matmul(m1,m2),dot_product(v1,v2)
```

Bit manipulations

Operate on the bits of integers (0,...,31 for 4-byte integer)

Single-bit functions (b=bit#):

`btest(i,b)` - `.true.` or `.false.`

`ibset(i,b),ibclr(i,b)` - `integer`

All-bit functions (pair-wise on two integers):

`iand(i,j),ior(i,j),ieor(i,j)` - `integer`

```
function bits(int)
integer :: i,int
character(32) :: bits
bits='00000000000000000000000000000000'
do i=0,31
  if (btest(int,i)) bits(32-i:32-i)='1'
enddo
end function bits
```

Processor time subroutine

`cpu_time(t)  -  t` = seconds after start of execution

```
integer :: i,nloop
real(8) :: sum
real    :: time0,time1

print*,'Number of operations in each loop’
read*,nloop

sum=0.0d0; call cpu_time(time0)
do i=1,nloop
  sum=sum+dfloat(i)*dfloat(i)
enddo
call cpu_time(time1)
print*,'Time used for s=s+i*i: ',time1-time0
```

Files

- A file has a name on disk, associated unit number in program
- File "connected" by `open` statement

```
open(unit=10,file='a.dat')
```

associates unit 10 with file a.dat

```
open(10,file='a.dat')
```

"`unit`" does not have to be written out

```
open(10,file='a.dat',status='old')
```

'`old`' file already exists ('`new`' ,'`replace`')

```
open(10,file='a.dat',status='old',access='append')
```

to append existing file with new data

Reading and writing files:

```
read(10,*)a
write(10,*)b
```

Output formatting

```
aa(1)=1; aa(2)=10; aa(3)=100; aa(4)=1000
bb(1)=1.d0; bb(2)=1.d1; bb(3)=1.d2; bb(4)=1.d3
print'(4i5)',aa
write(*,'(4i5)')aa
write(*,10)aa
10 format(4i5)
print'(4i3)',aa
print'(a,i1,a,i2,a,i3)',' one:',aa(1),' ten:',aa(2)
print'(4f12.6)',bb
```

```
    1   10  100 1000
    1   10  100 1000
    1   10  100 1000
  1 10100***
 one:1 ten:10
    1.000000   10.000000  100.000000 1000.000000
```

Allocatable arrays

Mechanism to assign the size of an array when running the program (i.e., not fixed when compiling)

```
integer :: m,n
real(8), allocatable :: matr(:,:)

write(*,*)'Give matrix dimensions m,n: ';
read*,m,n
allocate(matr(m,n))
…

deallocate(matr)
```

To change the size of an already allocated array, it first has To be de-allocated, then allocated again.

Variable-sized arrays, interfaces, assumed-shape, and automatic

```
integer :: m,n
real(8), allocatable :: matr(:,:)

Interface                               ! Declaring the interface
  subroutine checkmatr(matr)            ! of a procedure (include in
  real(8) :: matr(:,:)                  ! all procedures that need it,
  end subroutine checkmatr              ! e.g., when using “assumed shape”
end interface

write(*,*)'Give matrix dimensions m,n: '; read*,m,n
allocate(matr(m,n))
call checkmatr(matr)
end

subroutine checkmatr(matr)
real(8) :: matr(:,:)    ! Assumed shape
real(8) :: localmatr(size(matr,1),size(matr,2)) ! “Automatic”
print*,size(localmatr)
print*,shape(localmatr)
end subroutine checkmatr
```

Random number generators

How can deterministic algorithms give random numbers?

➢ pseudo-random numbers generators

Linear congruential generators; recurrence relation

$$x_{n+1} = \mathrm{mod}(a \cdot x_n + c, m)$$

can generate all numbers 0,...,m-1 in seemingly random order (for suitable a, m, c odd). Test with small $m=2^k$, c=1:

m=4	a	sequence:
	1	0 1 2 3 0
	2	0 1 3 3 3
	3	0 1 0 1 0
	4	0 1 1 1 1

m=8	a	sequence:
	1	0 1 2 3 4 5 6 7 0
	2	0 1 3 7 7 7 7 7 7
	3	0 1 4 5 0 1 4 5 0
	4	0 1 5 5 5 5 5 5 5
	5	0 1 6 7 4 5 2 3 0
	6	0 1 7 3 3 3 3 3 3
	7	0 1 0 1 0 1 0 1 0
	8	0 1 1 1 1 1 1 1 1

m=16	a	sequence:
	3	0 1 4 13 8 9 12 5 0 1 4 13 8 9 12 5 0
	5	0 1 6 15 12 13 2 11 8 9 14 7 4 5 10 3 0
	11	0 1 12 5 8 9 4 13 0 1 12 5 8 9 4 13 0
	13	0 1 14 7 12 13 10 3 8 9 6 15 4 5 2 11 0

On the computer, integer overflow is a modified modulus 2^{32} (or 2^{64}) operation; can be used for random numbers:

```
n=69069*n+1013904243
ran=0.5d0+n*0.23283064d-9
```

Many systems use this type of intrinsic random number generator

- don't use in serious work (period too short, not random enough)
- 64-bit integer version is quite a good generator

This one is recommended:

```
n=2862933555777941757*n+1013904243
ran=0.5d0+dble(n)*dmul
```

Where `dmul` is precalculated as

```
dmul=1.d0/dble(2*(2_8**62-1)+1)
```

Addition, subtraction can also be used, e.g.,

$$x_{n+1} = \mathrm{mod}(x_{n-3} - x_{n-1}, m) \qquad m = 2^k - \text{prime}$$

Mixed generators; longer periods, more random, e.g.,

```
mzran=iir-kkr
if (mzran < 0) mzran=mzran+2147483579
iir=jjr; jjr=kkr; kkr=mzran
nnr=69069*nnr+1013904243
mzran=mzran+nnr
rand=0.5d0+mzran*0.23283064d-9
```

Four seeds; `iir,jjr,kkr,nnr.`   Period > 10^{28}

More Fortran: Keyword and optional arguments

If the interface of the procedure is explicit (e.g., in a module)

- one does not have to use all arguments in a procedure call
  - omissions at the end of the argument list ok
  - any omission ok if keyword (dummy variable name) is used
- one can use any order of the arguments if keywords are used

Example: keyword.f90

```
module test
 contains
    subroutine keywordsub(a,b)
    integer, optional :: a
    integer, optional :: b
    if (present(a)) write(*,*)'a = ',a
    if (present(b)) write(*,*)'b = ',b
    end subroutine keywordsub
 end module test
```

```
program testkeyword
use test
integer :: arg1,arg2

read(*,*)arg1,arg2

write(*,*)
call keywordsub(arg1)
write(*,*)
call keywordsub(b=arg2)
write(*,*)
call keywordsub(a=arg1,b=arg2)

end program testkeyword
```

If arg1=1 and arg2=2 are read in, this is the output:

```
a=1

b=2

a=1
b=2
```

Fortran 90 intrinsic random number generator

`random_number(r)` initialized with `random_seed()`

```
integer :: i,size
integer, allocatable :: seed(:)
real :: r

call random_seed(size)
allocate (seed(size))
write(*,*)'give ',size,' random seeds '
read(*,*)seed

call random_seed(put=seed)
do i=1,10
   call random_number(r)
   write(*,*)r
end do

call random_seed(get=seed)
write(*,*)seed
```