Computational Techniques I
Course zero

ALFREDO M.J. SÁNCHEZ DE MERÁS
(WITH MATERIAL FROM PREVIOUS COURSES
BY J.MUGICA, J.M.HERMIDA AND A. PALACIOS)

Outline

1. Introduction
2. Data types
3. Declaration statements
4. Operators
5. Arrays
6. Program control: IF statements and DO loops
7. Formatted variable
8. I/O statements
9. Indexed variables
10. Procedures
11. Libraries
12. Other statements
13. Makefiles

Introduction

(Design and execution of a computer code)

High-level programming languages

- Computers work in binary system (machine language)
- Initially, numerical codes were used to represent operations such as "add the number in register A to the number in register B"
- By the beginning of 50's it was possible to use sets of keys to replace the numerical code for expressions like "RCL A; SUM B" (assembler)
- Clearly some kind of programming language which allowed a series of calculations to be expressed in something resembling mathematical notation was required
- In 1956 the first manual of the IBM Mathematical FORmula TRANslating System appears by John Backus *et al.* (high-level language)
- With it, also a translation program that converted the "text" into the numerical codes which the computer understood (compiler)

FORTRAN standards

FORTRAN (I-V, 66)

⇓

FORTRAN 77 (A lot of extensions: allocate, case-insensitive…)

⇓

FORTRAN 90/95

Extensions standarization
Pointers, modules
Matrix features

⇓

FORTRAN 2003 (Object-oriented, C-interoperability…)

⇓

FORTRAN 2018 (More strict rules)

Format of the fortran statements

FREE SOURCE FORMAT (f90 style, file.f90)

no limitation on placing in the line.

& at the end of the line for continuation.

Blanks are significant

! commented the line

Format of the fortran statements

FIXED SOURCE FORMAT (f77 style, file.f)

c: commented the line

Each column in each line counts!:

Column 1-5 are used for statement label number (usually blank)

Column 6: used ONLY to indicate line continuation (if non-blank character)

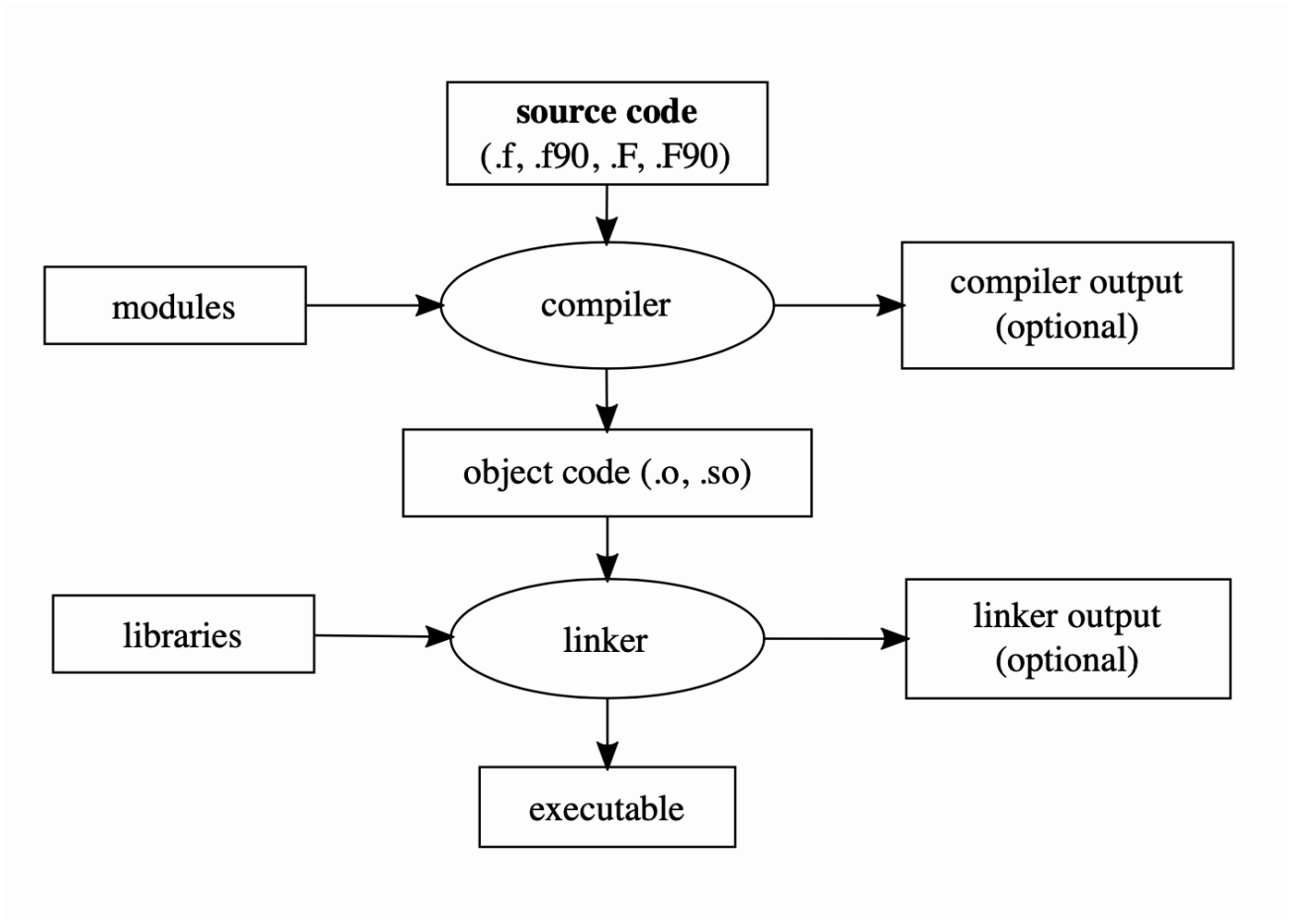
Column 7-72: fortran staments.; Column 73-80: ignored

For continuation*: & (or number) at column 6 in the next line.

Possible to indicate some lines of optional compilation (a D in the 1st column)

Several instructions in the same line (separated by ;)*

Running FORTRAN



Structured Programs

The design of a program has various phases:

- Definition of the problem ➔ Input data and results
- "*top-down*" design of the program ➔ Divide the program in modules and files.
- Design of each module ➔ Selection of the best algorithm and connection with the rest.

Basic goal: divide the problem into as many basic tasks as possible.

Algorithms and program flow

- Selection of algorithm:
  - Mathematical or numeric reasons
  - Computational

- Structure of an algorithm:

Data "*input*"
⇓
Data processing
⇓
"*output*" of the results

- Type of logical structures:
  - Execution control
  - Asignment (=)
  - Decision (IF)
  - Repetition (DO)

The structure of a fortran program

```
!————————————— Declaration area ————————————–
PROGRAM Example
! Declaration statements
!
IMPLICIT NONE
!
INTEGER :: s                          ! integer variables
REAL :: a, b, sum, product, power     ! real variables
!
! ————————————— Execution area ————————————–
s = 2
a = 4.
b = 6.
sum = a + b
product = a*b
power = a**s
!
WRITE(*,*) sum, product, power
! ————————————— End area ————————————–
!
STOP
END PROGRAM Example
! ————————————————————————————————————–
```

Data types

Declaration of variables

```
!————————————— Declaration area ——————————————
PROGRAM Example
! Declaration statements
!
IMPLICIT NONE
!
INTEGER :: s                          ! integer variables
REAL :: a, b, sum, product, power     ! real variables
!
! ————————————— Execution area ——————————————
s = 2
a = 4.
b = 6.
sum = a + b
product = a*b
power = a**s
!
WRITE(*,*) sum, product, power
! ————————————— End area ——————————————
!
STOP
END PROGRAM Example
! ————————————————————————————————————————
```

Data types

- A data type can be constant or variable.
- A *data type* implies:
  - A *finite* set of values
  - A way to represent them
  - A set of allowed operations among them
- There are *five* *basic intrinsic data types* → defined in fortran by default
- There can be *defined* *derived data types* → defined by the programmer

- Intrinsic Data Types:
  - Numeric: Integer, Real and Complex
  - Non-numeric: Character and logical

Memory storage. Units

- **bit** : Minimum unit to represent information (binary + digit : bit)
- **byte**: a group of 8 bits, that constitutes the minimum unit of storage in memory
- **kilobyte** (kB) : 1024 bytes
- **megabyte** (MB) : set of 1024x1024 bytes
- **gigabyte** (GB) : set of 1024x1024x1024 bytes
- **terabyte** (TB): 1024 GB

Memory storage. Variables

- **Integer Numbers**: they imply a translation from decimal -> binary. Exact representation, but in practice, there are limits to the magnitude of the integer number we can represent.
- **Real numbers**: represented by a mantissa + exponent. For instance, $110.5 = +0.1105 \times 10^3$. Both are represented as integer numbers. In general, representation not exact, as we can not represent a mantissa of infinite digits.
- **Complex numbers**: represented as a pair of real numbers (Re, Im).
- **Alphanumeric characters**:They are stored character by character as real numbers following the ASCII code.
- **Logical variables**: They are stored in one bit; Its value (.TRUE. or .FALSE.) is identified by 0 or 1.

Memory usage. The KIND attribute

The amount of memory stored for a variable can be specified with KIND:

KIND=n (n=1,2,4,8)

where *n* is the number of bytes (#bits = 8 n) stored.

Integer variables

- Declaration of an integer variable in the “Declaration area”:

```
INTEGER :: s
INTEGER([KIND=]n) :: s
INTEGER*n :: s
```

- Frequent subtypes ***n***={1,2,4,8}.
  - O.S. of 32 bits ➔ maximum *n=4*
  - O.S. of 64 bits ➔ maximum *n=8*

- The range of integer numbers are: -2^{m-1} to $2^{m-1}-1$ (m= 8 n)

  n=1 -128 to 127 (using ”two’s complement”)

  n=2 -32768 to 32767

✓ s=1 ; s=0 ; s=-33 ; s=+33 ; s=-67_2 (equivalent to KIND=2)

✗ s=9999999999999999 ; s=3.14 ; s=32,767 ; s=33_3

Real variables

- Declaration of real variables in the “Declaration area”:

REAL :: variables
REAL([KIND=]n) :: variables (with n= 4,8 or 16)
REAL*n :: variables
DOUBLE PRECISION

REAL(KIND=4)
3.14159 ; 3.14159_4: 62. ; –.00127 ; +5.0E3 ; 2E–3_4

REAL(KIND=8) (or **DOUBLE PRECISION**):
123456789D+5 ; 123456789E+5_8 ; +2.7843D00 ; 2E200_8 ; 2.3_8

REAL(KIND=16) :
123456789Q4000 ; –1.23Q–400 ; +2.72Q0 ; 1.88_16

- Valid ranges: 10^{37}-10^{-37} (_4), 10^{307} -10^{-307} (_8) and 10^{4931}-10^{-4931} (_16)

Complex variables

- Declaration of complex variables in the “Declaration area”

COMPLEX :: c

COMPLEX([KIND=]n) :: c

COMPLEX*n :: c

DOUBLE COMPLEX

- Valid examples of **COMPLEX**(KIND=4) :

(1.7039,–1.70391) ; (44.36_4,–12.2E16_4) ; (+12739E3,0.) ; (1.,2.)

- Valid examples of **COMPLEX**(KIND=16) :

(1.7039,–1.7039Q2) ; (547.3E0_16,–1.44) ; (+12739Q3,0.Q0)

Logical variables

- Declaration of logical variables in the “Declaration area”

LOGICAL :: l
LOGICAL([KIND=]*n*) :: l (with **n=**1,2,4,8)
LOGICAL*n :: l

- The logical constants are written as

.TRUE.[_*n*]
.FALSE.[_*n*]

Character variables

- Declaration of character variables in the “Declaration area”

CHARACTER :: a
CHARACTER([KIND=]*n*) :: a ; (with 0 ≤ *len* ≤ 2000, len being #characters)
CHARACTER([LEN=]*len*) :: a
CHARACTER([LEN=]*len* [,[KIND=]*n*]) :: a
CHARACTER(KIND=*n* [,LEN=*len*]) :: a
CHARACTER*len[,] :: a

- Valid examples:

"Orthogonal Vectors "
'YESTERDAY”S RESULTS’
“This isn’t wrong"
' '

Derived data types

```
type atom
    character(len=2)   :: name
    Integer :: Z
    real :: mass
    real :: q
    real, dimension(1:3) :: coord
end type atom
```

- Therefore, we can now define variables of *atom* type in the following way:

```
type(atom) :: carbon
```

- Assigning values to these variables:

```
carbon = atom(  "C ", 6, 12.011, 0.35, (/0.0, 1.0, 2.0/))
```

- We can select an element, for instance the atomic number: carbon%Z

- These structures can be combined into new *derived data types*

```
type molecule
     integer :: natom
     type(atom), dimension(1:1000) :: at
end type molecule
```

Declaration statements

Default declaration

If variables are not declared as belonging to a specific type, they are implicitly declared by default as follows:

Variables starting with : I, J, K, L, M, N:

Integer (_4) type by default

Variables starting with : A to H or O to Z

Real (_4) type by default

IMPLICIT statement

However, it is recommended to declare all variables to be used in a program. Thus, to prevent defaults, the following statement is used

IMPLICIT NONE

An error will be produced if an invoked variable has not been declared

IMPLICIT statement

Sacrificing safeness in the sake of simplicity

IMPLICIT DOUBLE PRECISION (A-H,O-Z)

or even

IMPLICIT DOUBLE PRECISION (A-H,O-Y)

IMPLICIT LOGICAL Z

No error will be generated but no checking will be done

Parameter statement

It allows to give a name to a particular constant

```
! define and assign the speed of light and its square
REAL, PARAMETER :: C = 2.9979251E8,C2= C**2

! define π number
real*8, parameter :: pi = acos(-1.0d0)

! define "useful"  real numbers
real*8, parameter :: one = 1.0d0, two = 2.0d0

! not restricted to numbers
logical, parameter :: dbgprt = .true.
character(15), parameter:: separation = '---------------'
```

```
      program prog_1
!
      implicit none
!
      integer :: i
      character (len=12) :: hi
      character (len=20) :: hi2
      character (len=10) :: nam
!
!
      hi  = 'Hello, I am'
      hi2 = hi
      nam = "Alfredo"
!
!     1) Free format
!
      print*, hi,nam
      write(6,*) hi,nam
      print*,
!
      print*, hi2,nam
      write(6,*) hi2,nam
!
      write(6,*) ('  ',i=1,4)
!
!     2) Fixed format
!
      print '(2a)', hi,nam
      write(6,'(a,3x,a)') hi, nam
      print*,
!
      print '(2a)', hi2,nam
      write(6,'(a12,a)') hi2, nam
!
!
      stop
      end program prog_1
```

```
 Hello, I am Alfredo
 Hello, I am Alfredo

 Hello, I am         Alfredo
 Hello, I am         Alfredo

Hello, I am Alfredo
Hello, I am    Alfredo

Hello, I am         Alfredo
Hello, I am Alfredo
```

EXAMPLE 1

```
      program prog_2
!
      implicit none
!
      integer*4 :: i, num, fact, isum
      integer (kind = 8) :: bigfac
      real*8 :: xlim4, xlim8
      real*8, parameter :: one = 1.0d0, two = 2.0d0
!
      write(6,*) 'Number?'
      read(5,*) num
      write(6,*)
      write(6,*)
!
      isum   = 0
      fact   = 1
      bigfac = 1_8
!
      do i = 1,num
         isum = isum + i
         fact = fact * i
!
         bigfac = bigfac * i
         write(6,'(i3,2i25)') i,fact,bigfac
      end do
!
      write(6,*)
      write(6,100) 'isum   :', isum
      write(6,100) 'fact   :', fact
      write(6,100) 'bigfac :', bigfac
  100 format(a,i25)
!
      xlim4 = two**31 - one
      xlim8 = two**63 - one
      write(6,'(//,a,d15.4)') 'Maximum integer*4 number', xlim4
      write(6,'(a,d15.4)') 'Maximum integer*8 number', xlim8
!
      stop
      end program prog_2
```

```
 Number?
22


  1                   1                   1
  2                   2                   2
  3                   6                   6
  4                  24                  24
  5                 120                 120
  6                 720                 720
  7                5040                5040
  8               40320               40320
  9              362880              362880
 10             3628800             3628800
 11            39916800            39916800
 12           479001600           479001600
 13          1932053504          6227020800
 14          1278945280          87178291200
 15          2004310016        1307674368000
 16          2004189184       20922789888000
 17          -288522240      355687428096000
 18          -898433024     6402373705728000
 19           109641728   121645100408832000
 20         -2102132736  2432902008176640000
 21         -1195114496 -4249290049419214848
 22          -522715136 -1250660718674968576

isum   :                   253
fact   :             -522715136
bigfac :  -1250660718674968576


Maximum integer*4 number     0.2147D+10
Maximum integer*8 number     0.9223D+19
```

EXAMPLE 2

Exercises

1. Write a “Hello world” program and modify it to experiment with different data types and arithmetic operations
2. Fibonacci numbers are a sequence of integers defined by the recurrence relation $F_n=F_{n-1} + F_{n-2}$ with the initial values $F_0=0$ and $F_1=1$. Print out the first 20 Fibonacci numbers. Using trial and error, determine the last $F_n < 1000000$. Print out the first 100 Fibonacci numbers.
3. Use the Fibonacci numbers above to estimate the golden ratio

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots = \lim_{n\to\infty} \frac{F_{n+1}}{F_n}$$

4. Numerically verify that $\varphi^2 - \varphi = \varphi - 1/\varphi = 1$

Arrays

Defining arrays

There are several forms of defining arrays:

1. Indicating between parenthesis the corresponding dimensions (The maximum number of dimensions is 7)

   ```
   integer :: idx(23)
   real :: amat(4,6)
   real(kind=8) :: dpmatrix(3,4,5,6)
   character (len = 80) :: screen(24)
   ```

2. Using the dimension attribute

3. Using the dimension statement (In this case, the type is determined by default naming or previous implicit statement)

Defining arrays

There are several forms of defining arrays:

1. Indicating between parenthesis the corresponding dimensions (The maximum number of dimensions is 7)
2. Using the dimension attribute

```
integer , dimension(23) :: idx
real , dimension (4,6) :: amat
real(kind=8) , dimension (3,4,5,6) :: dpmatrix
character (len = 80), dimension (24) :: screen
```

3. Using the dimension statement (In this case, the type is determined by default naming or previous implicit statement)

Defining arrays

There are several forms of defining arrays:

1. Indicating between parenthesis the corresponding dimensions (The maximum number of dimensions is 7)
2. Using the dimension attribute
3. Using the dimension statement (In this case, the type is determined by default naming or previous implicit statement)

   dimension b(13), c(0:5)

   dimension mat(5,8)

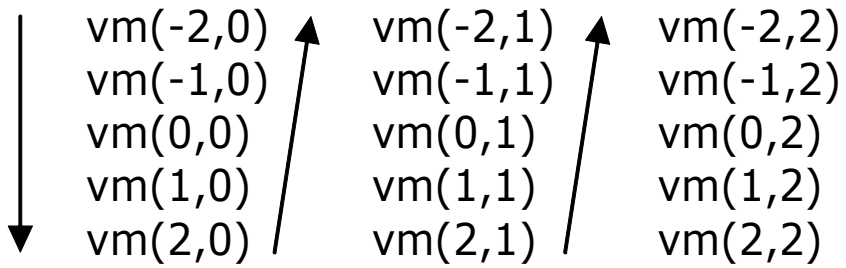
Indexing

- By default, indexing from 1 to n:

  ```
  real*8 vec(3)       ! Defines vec(1), vec(2), vec(3)
  real*8 vec(-1:1)    ! Defines vec(-1), vec(0), vec(1)
  ```

- Matrices are stored by columns (unlike C or Mathematica)

  ```
  real, dimension(-2:2,0:2) :: vm
  ```



Assigning arrays elements

- Elements:

```
vm(1,0) =1.
vm(j,k)  =0.                ! -2 ≤ j ≤ 2 & 0 ≤ k ≤ 2
B(5:7) = (/1.D0, 2.D0, 3.D0/)
vector(0:10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]       ! New standard
```

- *Sections*:

```
B(5:7)=1.D0          ! assigning 1.D0 to B(5),B(6) and B(7) elements
vm(-2:2,1)=0.        ! assigning zero to the second column
vm=0.                ! assigning zero to the whole matrix
vector(1:3) = (/ (2*i, i=1,5,2) /)       ! implicit cycle/loop with step = 2
lhs(1:3, 0:9) = rhs(-2:0, 20:29)         ! this is ok
lhs(1:2, 0:9) = rhs(-2:0, 20:29)         ! but this isn't
a(1:49:2, 2:50:2) = 4.0       ! elements in odd rows and even columns (start:end:step)
```

Array syntax

Less loops required:

```
integer :: m = 100, n = 200
real :: a(m,n), x(n), y(m)
integer :: i, j
!
y = 0.0
!
do j = 1, n
    y(:) = y(:) + a(:,j) * x(j)
end do
```

```
do j = 1, n
   do i = 1, m
      y(i) = y(i) + a(i,j) * x(j)
   end do
end do
```

Operators and intrinsic functions

Arithmetic operators

Operations are carried up in the same order than in standard Math

```
real :: x, y
integer :: i
!
i = 3
!
x = 2.0**(-i) ! power and negation: maximum priority
x = x*real(i) ! multiplication and type change : “medium” priority
y = x/2.0     ! division: “medium” priority
x = 2.0+y     ! Addition and substraction: minimum priority
```

Use parenthesis when needed
Be careful with integer division (e.g. 4/5 = 0)

Relational operators

Operator	Alternative (f77)	Description
==	.eq.	Tests for equality of two operands
/=	.ne.	Test for inequality of two operands
>	.gt.	Tests if left operand is strictly greater than right operand
<	.lt.	Tests if left operand is strictly less than right operand
>=	.ge.	Tests if left operand is greater than or equal to right operand
<=	.le.	Tests if left operand is less than or equal to right operand

Logical operators

Operator	Description
.and.	TRUE if both left and right operands are TRUE
.or.	TRUE if either left or right or both operands are TRUE
.not.	TRUE if right operand is FALSE
.eqv.	TRUE if left operand has same logical value as right operand
.neqv.	TRUE if left operand has the opposite logical value as right operand
.xor.	TRUE if only left or right but not both operands are TRUE

Numerical functions

ABS (A)	Absolute value		FLOOR (A)*	Greatest integer .le. A
AIMAG (Z)	Imaginary part of Z		INT (A [, KIND])	Conversion to integer
AINT (A)*	Truncation to whole nº		MAX (X, Y [,Z,...])	Maximum value
ANINT (A)*	Nearest whole number		MIN (A1, A2 [,..])	Minimum value
CEILING (A)*	Least integer .ge. A		MOD (A, P)	Remainder function
CMPLX (X [, Y])*	Conversion to complex		MODULO (A, P)	Modulo function
CONJG (Z)	Conjugate of Z		NINT (A [, KIND])	Nearest integer
DBLE (A)	Conversion to real *8		REAL (A [, KIND])	Conversion to real
DIM (X, Y)	Positive difference		SIGN (A, B)	Transfer of sign

(*) Optional KIND argument admitted (normally standard from 2003)

Mathematical functions

Function	Description		Function	Description
ACOS (X)	Arccosine		LOG (X)	Natural logarithm
ASIN (X)	Arcsine		LOG10 (X)	Base 10 logarithm
ATAN (X)	Arctangent		SIN (X)	Sine
ATAN2 (Y, X)	Arctangent		SINH (X)	Hyperbolic sine
COS (X)	Cosine		SQRT (X)	Square root
COSH (X)	Hyperbolic cosine		TAN (X)	Tangent
EXP (X)	Exponential		TANH (X)	Hyperbolic tangent

Archaic forms: DCOS(X), DASIN(X)…
Extensions: ERF(X), BESSEL_J1(X) (BESJ1(X))…

Character functions

Function	Description		Function	Description
ACHAR (I)	Character in ASCII		//	Concatenation operator
ADJUSTL (ST)	Adjust left		LGE (ST1,ST2)	Lexically .ge.
ADJUSTR (ST)	Adjust right		LGT (ST1,ST2)	Lexically .gt.
CHAR (I [, KIND])	Character in processor		LLE (ST1,ST2)	Lexically .le.
IACHAR (C)	ASCII code of C		LLT (ST1,ST2)	Lexically .lt.
ICHAR (C)	Processor code of C		REPEAT (ST, NC)	Repeated concatenation
INDEX (ST, SUBST [, BACK])	Starting position of a substring		SCAN (STRING, SET [, BACK])	Position of first match of STRING in SET
LEN (STRING)	Length of string		TRIM (STRING)	Remove trailing blanks
LEN_TRIM (ST)	Length without trailing blank characters		VERIFY (STRING, SET [, BACK])	Position of first non-match of STRING in SET

Control structures

Conditional structures (IF)

Do something or another depending on a condition, which may be from a simple comparison to a complex combination using logical operators. Two forms:

```
if (condition) … ! do something
        ----------------------------


Example:

if (a .ge. 0) b = sqrt(a)
```

```
if (condition) then
….          ! do something
else if (condition2) then
…..         ! or maybe this
else
…           ! or at least this
end if
```

Conditional structure (SELECT)

- Allows a variable to be tested for equality against a list of values
- The first match prevents further checking
- Normally with integer or character variables
- Each case block can contain multiple statements

```
      program selex
!
      implicit none
      character :: a ; integer :: icode
!
      write(6,*) 'Type in a character'
      read(5,*) a
!
!     Get code (code of '0' is 48; of '1', 49..)
!
      icode = ichar(a) - 48
      select case (icode)
         case (0,1,2,3,4)
            write(6,*) 'Number .lt. 5'
         case (5:9)
            write(6,*) 'Number .ge. 5'
         case default
            write(6,*) 'Not a number'
      end select
!
      stop     ! Actually, not needed'
      end
```

Repetive structures (DO)

Three loop formats available in Fortran:

1. Integer counter (fixed number of iterations)
2. Condition controlled (do until condition is false)
3. Explicit exit statement

```
do {control clause}
    …. ! execute something again and again until stopped
end do
```

Repetive structures (DO)

Three loop formats available in Fortran:

1. Integer counter (fixed number of iterations)
2. Condition controlled (do until condition is false)
3. Explicit exit statement

```
a = 0.0_8
do i = 0,20,2
   a = a + real(i,8)
end do
```

```
      a = 0.0d0
      do 100 i = 0,20,2
            a = a + real(i,8)
100 continue
```

Repetive structures (DO)

Three loop formats available in Fortran:

1. Integer counter (fixed number of iterations)
2. Condition controlled (do until condition is false)
3. Explicit exit statement

```
i = 0
do while (i <= 100)
    if (i == 13) cycle    ! Goes to the end of the loop (13 will not be included)
    i = i + 1
end do
```

Repetive structures (DO)

Three loop formats available in Fortran:

1. Integer counter (fixed number of iterations)
2. Condition controlled (do until condition is false)
3. Explicit exit statement

```
i = 0
do
   i = i + 1
   if (i > 100) exit        ! Leaves the loop
end do
```

Construct names

- We can put a name to control structures (IF, DO, SELECT)

- It helps in long structures
- cycle or exit can refer to name (for e.g. exit the outer loop from the inner one)
- Alternatively (not reccomended!) use label style

```
program loop
!
implicit none
integer :: i,j,ij
real, dimension(10,10) :: amat
!
!
amat = 0.0
!
do i =1,10
   do j = 1,10
      ij = 10*i + j
      amat(i,j) = real(ij)
   end do
end do
!
write(6,'(/a/)') 'The first matrix'
do i = 1,10
   write(6,'(10f8.1)') (amat(i,j), j=1,10)
end do
!
amat = 0.0
!
```

DO EXAMPLE (1/4)

```
!
loop1 : do i = 1,10
  loop2 :  do j = 1,10
              ij = 10*i + j
        test : if (ij == 13) then
                  cycle
               else if (ij == 25) then
                  cycle loop1
               else if (ij == 33) then
                  exit
               else if (ij == 68) then
                  exit loop1
               else
                  amat(i,j) = real(ij)
               end if test
            end do loop2
         end do loop1
!
write(6,'(/a/)') 'The second matrix'
do i = 1,10
   write(6,'(10f8.1)') (amat(i,j), j=1,10)
end do
!
end program loop
```

```
The first matrix

    11.0    12.0    13.0    14.0    15.0    16.0    17.0    18.0    19.0    20.0
    21.0    22.0    23.0    24.0    25.0    26.0    27.0    28.0    29.0    30.0
    31.0    32.0    33.0    34.0    35.0    36.0    37.0    38.0    39.0    40.0
    41.0    42.0    43.0    44.0    45.0    46.0    47.0    48.0    49.0    50.0
    51.0    52.0    53.0    54.0    55.0    56.0    57.0    58.0    59.0    60.0
    61.0    62.0    63.0    64.0    65.0    66.0    67.0    68.0    69.0    70.0
    71.0    72.0    73.0    74.0    75.0    76.0    77.0    78.0    79.0    80.0
    81.0    82.0    83.0    84.0    85.0    86.0    87.0    88.0    89.0    90.0
    91.0    92.0    93.0    94.0    95.0    96.0    97.0    98.0    99.0   100.0
   101.0   102.0   103.0   104.0   105.0   106.0   107.0   108.0   109.0   110.0
```

```
The second matrix

    11.0    12.0     0.0    14.0    15.0    16.0    17.0    18.0    19.0    20.0
    21.0    22.0    23.0    24.0     0.0     0.0     0.0     0.0     0.0     0.0
    31.0    32.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
    41.0    42.0    43.0    44.0    45.0    46.0    47.0    48.0    49.0    50.0
    51.0    52.0    53.0    54.0    55.0    56.0    57.0    58.0    59.0    60.0
    61.0    62.0    63.0    64.0    65.0    66.0    67.0     0.0     0.0     0.0
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
```

Exercises

5. Write a simple program to explore INT, NINT, FLOOR,...
6. Repeat exercise 2 using arrays and proper control structures
7. Interpret the outputs of the do example.
8. Cholesky decomposition allows to write a positive definite matrix as

$$\mathbb{M} = \boldsymbol{L}\,\boldsymbol{L^T} \longrightarrow M_{pq} = \sum_{J=1}^{n} L_p^J\, L_q^J \quad \text{with } L_i^J = \frac{M_{i,J}^{J-1}}{\sqrt{M_{J,J}^{J-1}}}$$

Construct a positive definite 5×5 matrix and Cholesky-decompose it

Input and output

Formatted and unformatted files

Fortran files are identified by numbers

By default, almost all fortran files are not human-readable (they are unformatted), the only exceptions being units 5 (input) and 6 (output).

As already seen, the two basic commands for I/O are READ and WRITE. Their syntax is the same:

READ(file[,format]) var1 [,var2,var3…]

WRITE(file [,format]) [var1,var2,var3…]

(an empty line is written if no variable is given)

Format and edit descriptors

The format consists in a set of edit descriptors

The format can be given either as a label either as a character variable or constant:

```
      WRITE(6,100) ivar1,rvar1,rvar2,rvar3,rvar4
100 FORMAT(I3,4F15.5)
```

```
FMT = '(I3,4F15.5)'
WRITE(6,FMT) ivar1,rvar1,rvar2,rvar3,rvar4
```

```
WRITE(6, '(I3,4F15.5)') ivar1,rvar1,rvar2,rvar3,rvar4
```

Format and edit descriptors

Some possible edit descriptors are:

- Iw w positions for an integer number (variable or constant)
- Fw.d w positions for a real number with d decimal digits
- Ew.d w positions with d decimal digits in scientific notation
- Dw.d w positions with d decimal digits in scientific notation
- Gw.d Fw.d or Ew.d as convenient
- Aw w positions for characters
- Xw w empty positions

Other I/O statements

- Files others than 5 or 6 should be explicitly opened:

  open([unit=]n [,file=..] [,status=..],[,...])
  - file = 'filename'
  - status = {'old', 'new', 'replace', 'scratch', 'unknown'}
  - access = {'sequential', 'direct'}
  - form = {'formatted', 'unformatted'}
  - recl = 'record length'
- Once a file is not going to be used, it should be closed:

  close ([unit=]n [,status={'keep','delete')
- If a file is to be read/written from its beginning:

  rewind ([unit=]n)
- To go back a record

  backspace ([unit=]n)

Arrays intrinsic functions

Arrays: functions and manipulations

Since Fortran90 there are a lot of functions to work with matrices

They allow for

- Determining dimension, number of elements…
- Finding specific numerical values
- Manipulating matrices
- Carrying up mathematical operations

Arrays: functions and manipulations

We will review them with an example

```
      program arrays
!
      implicit none
!
      real(kind=8) :: xmat(-1:1,5) , vec(-7:7), vmat(3,5),
     &                bmat(5,5), tmat(5,3), zmat(3,3),
     &                pad_dat(7)
      real(kind=8), parameter :: two = 2.0d0, one = 1.0d0, half = 0.5d0
      integer(kind=4) :: i, j
      character(len=15), parameter :: fmti ='(a20,5i5)',
     &                                fmtr ='(a20,5f12.5)'
!
      vec = (/ (real(i,8), i=-7,7) /)    ! Fortran 2003 [...] == (/.../)
      pad_dat = (/ (real(i,8)+half, i=1,7) /)
      call random_number(xmat)
      xmat = xmat - half
!
```

```
      write(6,'(/,a,7f10.3,/,8f10.3,/)') 'The vector',
     &         (vec(j), j=-7,-1), (vec(j), j=0,7)
!
      write(6,fmti) 'size vec :',size(vec),size(vec,1)
      write(6,fmti) 'shape vec :',shape(vec)
      write(6,fmti) 'lbound vec :', lbound(vec)
      write(6,fmti) 'ubound vec :', ubound(vec)
      write(6,*)
      write(6,fmtr) 'minval vec :',minval(vec),minval(vec,1),
     &                             minval(vec,mask=abs(vec)<3.5)
      write(6,fmti) 'minloc vec :',minloc(vec),minloc(vec,1),
     &                             minloc(vec,mask=abs(vec)<3.5)
      write(6,*)
```

```
The vector    -7.000    -6.000    -5.000    -4.000    -3.000    -2.000    -1.000
     0.000     1.000     2.000     3.000     4.000     5.000     6.000     7.000

         size vec :    15   15
        shape vec :    15
       lbound vec :    -7
       ubound vec :     7

       minval vec :     -7.00000     -7.00000     -3.00000
       minloc vec :     1    1    5
```

```
integer :: a (-1:1,4)    ! Equivalent to a(-1:1,1:4)
```

$$\mathbb{A} = \begin{pmatrix} 8 & 0 & -3 & -4 \\ -1 & 2 & 6 & 5 \\ 9 & 5 & 3 & -7 \end{pmatrix}$$

```
shape(a)     ⟶   3   4        ! 3 rows and 4 columns
size(a)      ⟶  12            ! 12 elements in total
size(a,1)    ⟶   3            ! 3 rows
size(a,2)    ⟶   4            ! 4 columns
lbound(a)    ⟶  -1   1        ! indexing of rows starts at -1; columns at 1
lbound(a,1)  ⟶  -1            ! indexing of rows starts at -1
lbound(a,2)  ⟶   1            ! indexing of columns starts at 1
ubound(a)    ⟶   1   4        ! indexing of rows ends at 1; columns at 4
ubound(a,1)  ⟶   1            ! indexing of rows ends at 1
ubound(a,2)  ⟶ 4              ! indexing of columns ends at 4
```

In the case of matrices, each dimension can be analyzed

```
      write(6,*) 'The matrix'
      do i = -1,1
         write(6,'(5f12.6)') (xmat(i,j), j=1,5)
      end do
      write(6,*)
!
      write(6,fmti) 'shape xmat :',shape(xmat)
      write(6,fmti) 'size xmat :',size(xmat),size(xmat,1),size(xmat,2)
      write(6,fmti) 'lbound xmat :', lbound(xmat)
      write(6,fmti) 'lbound xmat(dim1) :', lbound(xmat,1)
      write(6,fmti) 'lbound xmat(dim2) :', lbound(xmat,2)
      write(6,fmti) 'ubound xmat :', ubound(xmat),ubound(xmat,1),
     &                               ubound(xmat,2)

        shape xmat :    3    5
         size xmat :   15    3    5
       lbound xmat :   -1    1
 lbound xmat(dim1) :   -1
 lbound xmat(dim2) :    1
       ubound xmat :    1    5    1    5
```

```
integer :: a (-1:1,4)    ! Equivalent to a(-1:1,1:4)
```

$$\mathbb{A} = \begin{pmatrix} 8 & 0 & -3 & -4 \\ -1 & 2 & 6 & 5 \\ 9 & 5 & 3 & -7 \end{pmatrix}$$

```
maxval(a)     ⟶   9              ! Maximum value of matrix elements
maxval(a,mask=mod(xmat,2.0)==0.0)
              ⟶   8              ! Maximum value of even matrix elements
maxval(a,1)   ⟶   9  5  6  5     ! Maximum value of each column
maxval(a,2 )  ⟶   8  6  9        ! Maximum value of each row
maxloc(a)     ⟶   3  1           ! The indices of maxval(a)
maxloc(a,1)   ⟶   3  3  2  2     ! In which rows are maxval of each column
maxloc(a,2)   ⟶   1  3  1        ! In which columns are maxval of each row
```

```
!
      write(6,fmtr) 'minval xmat :',minval(xmat),
     &                              minval(xmat,mask=xmat>0.0)
      write(6,fmtr) 'minval xmat_col :',minval(xmat,1)
      write(6,fmtr) 'maxval xmat_row :',maxval(xmat,2)
      write(6,fmti) 'minloc xmat :',minloc(xmat)
      write(6,fmti) 'minloc xmat_col :',minloc(xmat,1)
      write(6,fmti) 'maxloc xmat_row :',maxloc(xmat,2)
      write(6,*)
!
 The matrix
   -0.027667    0.240682    0.010644   -0.112907   -0.336583
   -0.040477    0.406349   -0.104848   -0.416482   -0.053095
   -0.067364    0.372761   -0.049117    0.101061   -0.055366

       minval xmat :    -0.41648     0.01064
   minval xmat_col :    -0.06736     0.24068    -0.10485    -0.41648    -0.33658
   maxval xmat_row :     0.24068     0.40635     0.37276
       minloc xmat :    2    4
   minloc xmat_col :    3    1    2    2    1
   maxloc xmat_row :    2    2    2
```

```
integer :: a (-1:1,4)    ! Equivalent to a(-1:1,1:4)
```

$$\mathbb{A} = \begin{pmatrix} 8 & 0 & -3 & -4 \\ -1 & 2 & 6 & 5 \\ 9 & 5 & 3 & -7 \end{pmatrix}$$

```
sum(a)          ⟶  23          ! Sum of all matrix elements
sum(a, a.gt.0)  ⟶  38          ! Sum of positive matrix elements
sum(a,dim=1)    ⟶  16  7  6  -6 ! Sum of each column
sum(a,dim=2)    ⟶  1  12  10    ! Sum of each row
sum(a(-1:0,3:4)) ⟶  4          ! Sum of the red submatrix elements
```

```
!
      write(6,'(a20,3f17.1)') 'sum vec elements :',sum(vec),
     &                                    sum(vec,vec.gt.0)
      write(6,'(a20,3f17.1)') 'product vec ele. :',product(vec),
     &                                    product(vec,vec.ne.0),
     &                                    product(vec,vec.gt.0)
      write(6,*)
      write(6,fmtr) 'sum xmat elements :',sum(xmat),
     &                                    sum(xmat,xmat.gt.0)
      write(6,fmtr) 'sum xmat_col ele. :',sum(xmat,dim=1)
      write(6,fmtr) 'sum xmat_row ele. :',sum(xmat,dim=2)
      write(6,fmtr) 'sum xmat section :',sum(xmat(-1:0,4:5))
      write(6,*)
```

```
The matrix
  -0.027667    0.240682    0.010644   -0.112907   -0.336583
  -0.040477    0.406349   -0.104848   -0.416482   -0.053095
  -0.067364    0.372761   -0.049117    0.101061   -0.055366

 sum vec elements :              0.0            28.0
 product vec ele. :             -0.0      -25401600.0          5040.0

sum xmat elements :    -0.13241     1.13150
sum xmat_col ele. :    -0.13551     1.01979    -0.14332    -0.42833    -0.44504
sum xmat_row ele. :    -0.22583    -0.20855     0.30198
 sum xmat section :    -0.91907
```

```
The vector    -7.000    -6.000    -5.000    -4.000    -3.000    -2.000    -1.000
     0.000     1.000     2.000     3.000     4.000     5.000     6.000     7.000
```

```
      vmat = reshape(vec,(/3,5/))
      do i = 1,3 ; write(6,'(5f12.6)') (vmat(i,j), j=1,5) ; end do
      write(6,*)
!
      vmat = reshape(vec,(/3,5/),order=(/1,2/))
      do i = 1,3 ; write(6,'(5f12.6)') (vmat(i,j), j=1,5) ; end do
      write(6,*)
!
      vmat = reshape(vec,(/3,5/),order=(/2,1/))
      do i = 1,3 ; write(6,'(5f12.6)') (vmat(i,j), j=1,5) ; end do
  -7.000000   -4.000000   -1.000000    2.000000    5.000000
  -6.000000   -3.000000    0.000000    3.000000    6.000000
  -5.000000   -2.000000    1.000000    4.000000    7.000000

  -7.000000   -4.000000   -1.000000    2.000000    5.000000
  -6.000000   -3.000000    0.000000    3.000000    6.000000
  -5.000000   -2.000000    1.000000    4.000000    7.000000

  -7.000000   -6.000000   -5.000000   -4.000000   -3.000000
  -2.000000   -1.000000    0.000000    1.000000    2.000000
   3.000000    4.000000    5.000000    6.000000    7.000000
```

ARRAYS EXAMPLES: MANIPULATIONS (1/3)

```
The vector    -7.000    -6.000    -5.000    -4.000    -3.000    -2.000    -1.000
     0.000     1.000     2.000     3.000     4.000     5.000     6.000     7.000
```

```
      bmat = reshape(vec,(/5,5/),pad=pad_dat)
      write(6,fmtr) 'pad_dat(1:5) :',(pad_dat(i),i=1,5)
      write(6,fmtr) 'pad_dat(6:7) :',(pad_dat(i),i=6,7)
      write(6,*)
      do i = 1,5 ; write(6,'(5f12.6)') (bmat(i,j), j=1,5) ; end do
      write(6,*)
!

      pad_dat(1:5) :     1.50000     2.50000     3.50000     4.50000     5.50000
      pad_dat(6:7) :     6.50000     7.50000

   -7.000000   -2.000000    3.000000    1.500000    6.500000
   -6.000000   -1.000000    4.000000    2.500000    7.500000
   -5.000000    0.000000    5.000000    3.500000    1.500000
   -4.000000    1.000000    6.000000    4.500000    2.500000
   -3.000000    2.000000    7.000000    5.500000    3.500000
```

```
!
    forall(i=1:3, j=1:5) vmat(i,j) = vec(3*(j-1)+i-8) / two
    do i = 1,3 ; write(6,'(5f12.6)') (vmat(i,j), j=1,5) ; end do
    write(6,*)
!
    where(vmat > 0) vmat = one / vmat
    do i = 1,3 ; write(6,'(5f12.6)') (vmat(i,j), j=1,5) ; end do
    write(6,*)


-3.500000   -2.000000   -0.500000    1.000000    2.500000
-3.000000   -1.500000    0.000000    1.500000    3.000000
-2.500000   -1.000000    0.500000    2.000000    3.500000

-3.500000   -2.000000   -0.500000    1.000000    0.400000
-3.000000   -1.500000    0.000000    0.666667    0.333333
-2.500000   -1.000000    2.000000    0.500000    0.285714
```

```
!     The previous forall using explicits loops
!
      write(6,'(3a3,2(3x,a4,3x))') ' i ',' j ',' k ',
     &                             'vec','vmat'
      do j = 1,5
         do i = 1,3
            k = 3*(j-1)+i-8     ! Just to be clear
            vmat(i,j) = vec(k) / two
            write(6,'(3i3,2f7.2)') i,j,k,vec(k),vmat(i,j)
         end do
      end do
!
```

i	j	k	vec	vmat
1	1	-7	-7.00	-3.50
2	1	-6	-6.00	-3.00
3	1	-5	-5.00	-2.50
1	2	-4	-4.00	-2.00
2	2	-3	-3.00	-1.50
3	2	-2	-2.00	-1.00
1	3	-1	-1.00	-0.50
2	3	0	0.00	0.00
3	3	1	1.00	0.50
1	4	2	2.00	1.00
2	4	3	3.00	1.50
3	4	4	4.00	2.00
1	5	5	5.00	2.50
2	5	6	6.00	3.00
3	5	7	7.00	3.50

The subsequent where statement takes only the strictly positive elements of vmat (those inside the red rectangle) an invert them

```
      write(6,fmtr) 'dot product vec·vec:',dot_product(vec,vec)
      write(6,*)
!
      vmat = reshape(vec,(/3,5/))
      xmat = one + vmat**two
      do i = -1,1
         write(6,'(5f12.6)') (xmat(i,j), j=1,5)
      end do
      write(6,*)
!
      tmat = transpose(vmat)
      do i = 1,5 ; write(6,'(5f12.6)') (tmat(i,j), j=1,3) ; end do
      write(6,*)
```

```
dot product vec·vec   280.00000
```

$$\left(\sum_{n=-7}^{7} n^2\right)$$

```
   50.000000   17.000000    2.000000    5.000000   26.000000
   37.000000   10.000000    1.000000   10.000000   37.000000
   26.000000    5.000000    2.000000   17.000000   50.000000

   -7.000000   -6.000000   -5.000000
   -4.000000   -3.000000   -2.000000
   -1.000000    0.000000    1.000000
    2.000000    3.000000    4.000000
    5.000000    6.000000    7.000000
```

```
!
      xmat = vmat + vmat
      do i = -1,1 ; write(6,'(5f12.6)') (xmat(i,j), j=1,5) ; end do
      write(6,*)
!
      xmat = vmat * vmat
      do i = -1,1 ; write(6,'(5f12.6)') (xmat(i,j), j=1,5) ; end do
      write(6,*)
!
  -14.000000   -8.000000   -2.000000    4.000000   10.000000
  -12.000000   -6.000000    0.000000    6.000000   12.000000
  -10.000000   -4.000000    2.000000    8.000000   14.000000

   49.000000   16.000000    1.000000    4.000000   25.000000
   36.000000    9.000000    0.000000    9.000000   36.000000
   25.000000    4.000000    1.000000   16.000000   49.000000
```

```
!
      zmat = matmul(vmat,transpose(vmat))
      do i = 1,3 ; write(6,'(5f12.6)') (zmat(i,j), j=1,3) ; end do
      write(6,*)
!
      bmat = matmul(transpose(vmat),vmat)
      do i = 1,5 ; write(6,'(5f12.6)') (bmat(i,j), j=1,5) ; end do
      write(6,*)
!
      end program arrays
```

```
  95.000000   90.000000   85.000000
  90.000000   90.000000   90.000000
  85.000000   90.000000   95.000000

 110.000000   56.000000    2.000000  -52.000000 -106.000000
  56.000000   29.000000    2.000000  -25.000000  -52.000000
   2.000000    2.000000    2.000000    2.000000    2.000000
 -52.000000  -25.000000    2.000000   29.000000   56.000000
-106.000000  -52.000000    2.000000   56.000000  110.000000
```

Dynamic allocation of arrays

Until now, arrays have been declared at compile time:

Dimensions are fixed!

(static allocation)

They can also be declared at run time by means of the statement ALLOCATE if previously defined with the attribute ALLOCATABLE:

Dimensions are not fixed!

(dynamic allocation)

```
!...
real*8, allocatable :: vec1(:), mat(:,:)
integer :: dimvec,nrow,ncol, ierr1, ierr2
!...
read(5,*) dimvec,nrow,ncol
!...
allocate(vec1(dimvec),stat=ierr1)
if (ierr1 /= 0) stop  ! Error
!...
allocate(mat(nrow,ncol),stat=ierr2)
if (ierr2 /= 0) stop  ! Error
!...
deallocate(vec1,stat=ierr1)
deallocate(mat,stat=ierr2)
if ((ierr1 /= 0) .or. (ierr2 /= 0)) stop
!...
```

Derived data types

```
type atom
    character(len=2)   :: name
    Integer :: Z
    real :: mass
    real :: q
    real, dimension(1:3) :: coord
end type atom
```

- Therefore, we can now define variables of *atom* type in the following way:

```
type(atom) :: carbon
```

- Assigning values to these variables:

```
carbon = atom(  "C ", 6, 12.011, 0.35, (/0.0, 1.0, 2.0/))
```

- We can select an element, for instance the atomic number: carbon%Z

- These structures can be combined into new *derived data types*

```
type molecule
     integer :: natom
     type(atom), dimension(1:1000) :: at
end type molecule
```

Exercises

9. Write a simple program to explore the different possibilities for formatting
10. Write a program that can write in a nice style a real matrix of 30 rows and 30 columns
11. Write simple programs to explore the different matrix functions.

Procedures

Modular programming

A program may (and frequently must) be split into smaller units (procedures).

This allows for:

- Faster compilation, as only new/modified units need to be (re)compiled
- Easier debugging, as errors are isolated
- Immediate reusability, as the same procedure can be used in different programs

In fortran there exist two types of procedures:

- Functions: Return a single value
- Subroutines: May return several values

Functions

[type] **function** &
fname(a1,a2..) [result(val)]

[declarations]

[statements]

[return] ! F77

end function fname

In the main program:

res = fname(v1,v2…)

```
real function side3_wres(a,b,gamm) result(c)
implicit none ; real :: a, b, gamm
c = sqrt(a**2 + b**2 - 2*a*b*cos(gamm)
end function side3_wres

! Or simply

real function side3(a,b,gamm)
implicit none ; real :: a, b, gamm
side3 = sqrt(a**2 + b**2 - 2*a*b*cos(gamm))
end function side3
!
!
program xxx
!...
x3 = side3(x1,x2,angle)
!...
end program xxx
```

Recursive functions

A function can be declared as recursive

Then the function calls itself

Of course, that implies that some initial value must be given

```
recursive function sumsqinv(n) result(res)
implicit none
real(kind=8) :: res, nsq ; integer :: n
   if (n <= 0) then
      stop 'n must be .gt. 0'
   else if (n == 1) then
      res = 1_8
   else
      nsq = real(n*n,8)
      res = 1_8/nsq + sumsqinv(n-1)
   end if
end function
```

Subroutines

subroutine &

subname(a1,a2,..)

[declarations]

[statements]

[return] ! F77

end subroutine fname

In the main program:

call subname(v1,v2…)

```
subroutine side3_sub(a,b,c,gamm,beta)
implicit none
real :: a, b, c, gamm, beta
!
c = sqrt(a**2 + b**2 - 2*a*b*cos(gamm))
beta=acos((a**2 + c**2 - b**2) / (2*a*c))
!
end subroutine side3_wres
!
!
!
program xxx
!...
call side3_sub(x1,x2,x3,ang_c,ang_b)
!...
end program xxx
```

Subroutine and function arguments

- Arguments are identified by their position in the arguments list, their names being irrelevant
- Actually, memory addresses and ranges are passed
- Unless if included in a module (or common in F77), variables not in the argument list are local to the procedure
- If a variable in the argument list is changed in the procedure, the main program inherits this change

The INTENT attribute

To avoid unintentional changes of variables in the argument list, the INTENT attribute can be used. There are three possibilities:

- INTENT(IN)

  Value can't be changed

- INTENT(OUT)

  Value must be given

- INTENT(INOUT)

  Can, but not must, change

```
subroutine sub1(xinp,xout,x1,x2)
!
real, intent(in) :: xinp
real, intent(out) :: xout
! No need to modufy x1 or x2
real, intent(inout) :: x1
real :: x2
intent(inout) :: x2
!....
xinp = 5.0  ! Error
!..
x1 = x2
!
! Some assigment for xout
!
xout = 3.1  ! Requiered
!...
end subroutine sub1
```

Types of procedures

Procedures can be classified as

- Intrinsic (fortran functions, e.g. LOG)
- Internal (declared in the same program unit by means of CONTAINS statement)
- Module (an external unit containing procedures, data definitions and/or variable declarations; defined by statement MODULE and called by statement USE)
- External (declared in a different program unit, e.g. a routine from a library)

Internal procedures: CONTAINS

Internal procedures (in the example the previous function sumsqinv) are inside the same program unit than the calling (sub)program after the statement CONTAINS

```
gfortran -std=f95 -o rec_exam.x rec_exam.f90
```

```
program rec_exam
implicit none
integer :: i ; real(kind=8) :: xsum
!
print*, 'Type in an integer' ; read(*,*) i
xsum = sumsqinv(i)
write(6,'(a,i4,a,g15.5)') 'Sum from 1 to', &
     i,' of 1/n^2 =',xsum
!
   contains
     recursive function sumsqinv(n) result(res)
     implicit none
     real(kind=8) :: res, nsq ; integer :: n
        if (n <= 0) then
           stop 'n must be .gt. 0'
        else if (n == 1) then
           res = 1_8
        else
           nsq = real(n*n,8)
           res = 1_8/nsq + sumsqinv(n-1)
        end if
     end function
end program
```

Modules

- Modules are external files that may contain procedures, data declarations and/or derived data definitions
- They are defined in modules blocks MODULE … END MODULE
- They can be called by used by other program units by means of the statement USE, maybe with the attribute ONLY
- Variables defined in the module can be PUBLIC (the default) or PRIVATE
- Modules must be compiled before linking main program

```
gfortran -c util.f   # Using F2003 iso_fortran_env
gfortran -c -std=f95 array_in_sub.f
gfortran -std=f95 -o array_in_sub.x util.o array_in_sub.o
```

```
      module util
!
      use, intrinsic :: iso_fortran_env, only: real64  ! F2003
      integer, private, parameter :: dp = real64
!
      integer :: i,j,k,l,m,n,p,q,r,s
      real(kind=dp), parameter :: zero = 0._8, one = 1._8, two = 2._8
      real(kind=dp), parameter :: pi = 3.14159265358979323846_8,
     &    hbar = 1.05457168e-34_8, toang = 0.5291772108_8,
     &    todeg = 180.0_8 / pi
!
      contains
         subroutine wrimat(xmat,nrow,ncol)
!
!        Subroutine to write a matrix in blocks of 5 columns
```

```
!
!       Subroutine to write a matrix in blocks of 5 columns
!
        implicit none
        integer :: nrow,ncol,istr, iend
        real(kind=8) :: xmat(nrow,ncol)
        integer, parameter :: maxcol = 5
!
        iend = 0
        do
            istr = iend + 1
            iend = iend + maxcol ; if (iend > ncol) iend = ncol
            do i = 1, nrow
               write(6,'(5g14.3)') (xmat(i,j),j=istr,iend)
            end do
            if (iend == ncol) exit
        end do
        return
        end subroutine
      end module
```

Arrays in procedures

The fact that only memory addresses and lengths are passed to procedures, allows for treating

- A vector as a matrix or vice versa
- Only a part of the array
- …

Note that this can be dangerous and, therefore, requires some care!!

```
      program fakedot
!
      use util ! to declare n and wrimat
      implicit none
      real(kind=8), dimension (2,2) :: a,b
      real(kind=8) :: res
!
      n = 2
      a(:,1) = (/ 1., 3. /) ; a(:,2) = (/ 2., 4. /)
      write(6,'(a,/)') 'Matrix A'
      call wrimat(a,n,n)
!
      b(:,1) = (/ 6., 8. /) ; b(:,2) = (/ 7., 9. /)
      write(6,'(//,a,/)') 'Matrix B'
      call wrimat(b,n,n)
!
```

```
c     Calling dot_product with matrices A and B produces an error
c     at compilation:
c     'vector_a' argument of 'dot_product' intrinsic at (1) must be of rank 1
!
!     res = dot_product(a,b)
!
c     Instead, this works nicely
!
      call cheat_dot(n,n,a,b,res)
      write(6,'(///,a,f6.1,/)') 'Sum_ij (A_ij * B_ij) =', res
c
      stop
!
      contains
        subroutine cheat_dot(nrow,ncol,a,b,c)
        implicit none
        real(kind=8), dimension(nrow*ncol) :: a,b
        real(kind=8) :: c
        integer :: nrow,ncol
        c = dot_product(a,b)
        return
        end subroutine
      end program fakedot
```

```
      program test_x
!
      use util
      implicit none
      real(kind=8), dimension(:,:) , allocatable :: a,b,c
      integer :: ierr
!
      allocate(a(3,3),b(5,3),c(5,3),stat=ierr)
      if (ierr /= 0) stop 'Error in allocation '
!
      do j = 1,3
         a(:,j) = (/ (3*(j-1)+i, i=1,3) /)
         b(:,j) = (/ (one / real(i+j,8), i=1,5) /)
      end do
!
      write(6,'(a,/)') 'In main before sub:'
      write(6,1) 'a: shape,lbound,ubound', shape(a),lbound(a),ubound(a)
      write(6,1) 'b: shape,lbound,ubound', shape(b),lbound(b),ubound(b)
      write(6,1) 'c: shape,lbound,ubound', shape(c),lbound(c),ubound(c)
      write(6,*)
    1 format(a,3(2i5,4x))
!
```

```
!
      m = 5 ; n = 3
      call sub(m,n,a,b,c)
!
      write(6,'(/,a,/)') 'In main after sub:'
      write(6,1) 'a: shape,lbound,ubound', shape(a),lbound(a),ubound(a)
      write(6,1) 'b: shape,lbound,ubound', shape(b),lbound(b),ubound(b)
      write(6,1) 'c: shape,lbound,ubound', shape(c),lbound(c),ubound(c)
      write(6,'(//,a,/)') 'Matrix c in main'
      call wrimat(c,m,n)
!
      contains
      subroutine sub(nrow,ncol,a,b,c)
!
      implicit none
      integer, intent(in) :: nrow,ncol
      integer :: i,j
      real(kind=8), dimension(:,:), intent(in) :: a,b
      real(kind=8), intent(out) :: c(-(nrow-1)/2:(nrow-1)/2,ncol)
      real(kind=8)  :: x(nrow,ncol)
!
```

Assumed-shapes arrays

Fitted-indexed array

Automatic array (local)

```
!
      write(6,'(//,a,/)') 'In sub:'
      write(6,1) 'a: shape,lbound,ubound', shape(a),lbound(a),ubound(a)
      write(6,1) 'b: shape,lbound,ubound', shape(b),lbound(b),ubound(b)
      write(6,1) 'c: shape,lbound,ubound', shape(c),lbound(c),ubound(c)
      write(6,1) 'x: shape,lbound,ubound', shape(x),lbound(x),ubound(x)
    1 format(a,3(2i5,4x))
!
      x = 2.0_8 / b
      c = matmul(x,a)
!
      return
      end subroutine
      end program
```

```
In main before sub:

a: shape,lbound,ubound    3    3        1    1        3    3
b: shape,lbound,ubound    5    3        1    1        5    3
c: shape,lbound,ubound    5    3        1    1        5    3



In sub:

a: shape,lbound,ubound    3    3        1    1        3    3
b: shape,lbound,ubound    5    3        1    1        5    3
c: shape,lbound,ubound    5    3       -2    1        2    3
x: shape,lbound,ubound    5    3        1    1        5    3
```

```
In main after sub:

a: shape,lbound,ubound    3    3        1    1        3    3
b: shape,lbound,ubound    5    3        1    1        5    3
c: shape,lbound,ubound    5    3        1    1        5    3


Matrix c in main

     40.0000          94.0000          148.000
     52.0000          124.000          196.000
     64.0000          154.000          244.000
     76.0000          184.000          292.000
     88.0000          214.000          340.000
```

Only the indexing of c varies:

- In main: c(5,3)
- In sub: c(-2:2,3)

But it refers always to same memory position

External procedures: INTERFACE

As memory addresses are passed, kind and ranges must match

The compiler "knows" the data types used by intrinsic, internal or module procedures. But not for external procedures!

INTERFACE provides such information in order the compiler to be able of checking for mismatches

```
program rec_exam
!
implicit none
integer :: i ; real(kind=8) :: xsum
!
interface
     recursive function sumsqinv(n) result(res)
     real(kind=8) :: res, nsq ; integer :: n
     end function
end interface
!
print*, 'Type in an integer' ; read(*,*) i
xsum = sumsqinv(i)
write(6,'(a,i4,a,g15.5)') 'Sum from 1 to', &
     i,' of 1/n^2 =',xsum
!
end program
```

```
gfortran -c -std=f95 recfunc.f90
gfortran -c -std=f95 recfunc_ext.f90
gfortran -o recfunc_ext.x -std=f95 recfunc.o recfunc_ext.o
```

INTERFACE: other use

INTERFACE can also be used to pass the name of a procedure as argument to other procedure

```
subroutine sub(namefun,x,…)
!
implicit none
!....
interface
    function namefunc(p,q)
       real*8 :: namefunc,p,q
    end function
end interface
```

```
      program newraph
c
      implicit none
      real(kind=8) :: x0, x1, diff, fun, der
      integer(kind=4) :: iter
      real(kind=8), parameter :: thres = 1.0d-8
c
      interface
         function f(x)
         real(kind=8) :: f,x
         end function
      end interface
c
      write(6,*) 'Initial guess?' ; read(5,*) x0
      write(6,'(/,a4,2(6x,a6,5x),4x,2(a7,7x))') "Iter","x_0",
     &                                 "f(x_0)","f'(x_0)","x_1"
c
      iter = 0
      do
          iter = iter + 1 ; fun  = f(x0)
          call mkder(f,der,x0)
          x1   = x0 - fun/der
          write(6,'(i3,4f16.8)') iter,x0,fun,der,x1
c
```

```
         diff = abs(x1-x0)
         if (diff .lt. thres) then
            exit
         else
            x0 = x1
         end if
      end do
      stop
c
      contains
c
         subroutine mkder(f,d,x)
         real(kind=8) d,x
         real(kind=8), parameter :: h = 1.0d-3, two = 2.0d0
c
           interface
              function f(x)
              real(kind=8) :: f,x
              end function
            end interface
c
         d = (f(x+h) - f(x-h)) / (two*h)
c
         return
         end subroutine
      end program
```

INTERFACE EXAMPLE (2/3)

The function is programmed in a separate file

```
function f(x)
real(kind=8) :: f,x
f = 4.0_8*x**3 - 0.1223_8*x*x + 0.1223_8*x - 0.03058_8
return
end function
```

To compile, link and run

```
[macold_556 % gfortran -c newraph.f
[macold_557 % gfortran -c funcion_f.f
[macold_558 % gfortran -o newraph.x funcion_f.o newraph.o
[macold_559 %
[macold_559 % ./newraph.x
 Initial guess?
0.1

 Iter          x_0            f(x_0)          f'(x_0)           x_1
   1       0.10000000     -0.01557300       0.21784400       0.17148694
   2       0.17148694      0.00696847       0.43325152       0.15540280
   3       0.15540280      0.00048414       0.37409285       0.15410862
   4       0.15410862      0.00000292       0.36960264       0.15410073
   5       0.15410073      0.00000000       0.36957540       0.15410073
[macold_560 %
```

Exercises

12. Convert your code to calculate the n^{th}-term of a Fibonacci series into a function
13. Convert your code to sum the first n^{th}-terms of the succession $1/n^2$ into a function to sum $1/n^k$, where k is an integer given also in the argument list
14. Convert your codes for matrix multiplication, matrix writing and Cholesky decomposition into three subroutines
15. Use to previous procedures to write a program that Cholesky decompose the matrix $\sum_{i=1}^{3} \overrightarrow{v_i}\,\overrightarrow{w_i}$, where $\overrightarrow{v_i}$ are column vectors built with elements calculated using Fibonacci series and $\overrightarrow{w_i}$ are row vectors built with elements calculated by summing successions of powers of inverse natural numbers. The output must look nice.

Libraries

Creating static libraries

Object files can be conveniently grouped in libraries.

Static libraries (lib.a)

- The executable contains a copy of the library.
- Then, larger size but faster
- Created and managed by means of ar utility
- Linked using options –l and –L

```
513 % ar -r libfunc.a function_*.o
514 % ar -t libfunc.a
__.SYMDEF
function_1.o
function_2.o
function_4.o
function_5.o
function_3.o
515 % ar -d libfunc.a function_3.o
516 % gfortran -o prog.x main.o -lfunc
```

Creating dynamic libraries

Object files can be conveniently grouped in libraries.

Dynamic libraries (lib.so)

- Accessed at run time through environment variable LD_LIBRARY_PATH
- Executable of smaller size, but slower
- Created using the compiler
- Examined by nm

```
132 % gfortran -c -fPIC function_*.f
133 % gfortran -shared -o libfunc.so function_*.o
134 % gfortran -o newraph.x newraph.o -L. -lfunc
135 % mv libfunc.so $HOME
136 % ./newraph.x
./newraph.x: error while loading shared libraries: libfunc.so:
cannot open shared object file: No such file or directory
137 % export LD_LIBRARY_PATH=$HOME:$LD_LIBRARY_PATH
138 % ./newraph.x
 Initial guess?
```

BLAS and LAPACK libraries

There exist pre-programmed libraries for different purposes. Among them, two very often used in Physics and Chemistry are

- BLAS libblas.a (libblas.so)

  https://www.netlib.org/blas/

  https://math-atlas.sourceforge.net/ (tunable implementation)

- LAPACK liblapack.a (liblapack.so)

  https://www.netlib.org/lapack/

gfortran file.o –L <*library_directory*> -lblas -llapack

BLAS level 1 routine DCOPY

- dcopy()

```
subroutine dcopy ( integer                          N,
                   double precision, dimension(*)   DX,
                   integer                          INCX,
                   double precision, dimension(*)   DY,
                   integer                          INCY
                 )
```

DCOPY

Purpose:

```
     DCOPY copies a vector, x, to a vector, y.
     uses unrolled loops for increments equal to 1.
```

BLAS level 1 routine DDOT

- ddot()

```
double precision function ddot ( integer                    N,
                                 double precision, dimension(*) DX,
                                 integer                    INCX,
                                 double precision, dimension(*) DY,
                                 integer                    INCY
                               )
```

DDOT

Purpose:

```
     DDOT forms the dot product of two vectors.
     uses unrolled loops for increments equal to one.
```

BLAS level 1 routine DSCAL

- dscal()

```
subroutine dscal ( integer                      N,
                   double precision             DA,
                   double precision, dimension(*) DX,
                   integer                      INCX
                 )
```

DSCAL

Purpose:

```
     DSCAL scales a vector by a constant.
     uses unrolled loops for increment equal to 1.
```

BLAS level 1 routine DAXPY

- daxpy()

```
subroutine daxpy ( integer                        N,
                   double precision               DA,
                   double precision, dimension(*) DX,
                   integer                        INCX,
                   double precision, dimension(*) DY,
                   integer                        INCY
                 )
```

DAXPY

Purpose:

```
     DAXPY constant times a vector plus a vector.
     uses unrolled loops for increments equal to one.
```

BLAS level 2 routine DGEMV (1/2)

DGEMV

Purpose:

```
DGEMV  performs one of the matrix-vector operations

   y := alpha*A*x + beta*y,   or   y := alpha*A**T*x + beta*y,

where alpha and beta are scalars, x and y are vectors and A is an
m by n matrix.
```

BLAS level 2 routine DGEMV (2/2)

- dgemv()

```
subroutine dgemv ( character                          TRANS,
                   integer                            M,
                   integer                            N,
                   double precision                   ALPHA,
                   double precision, dimension(lda,*) A,
                   integer                            LDA,
                   double precision, dimension(*)     X,
                   integer                            INCX,
                   double precision                   BETA,
                   double precision, dimension(*)     Y,
                   integer                            INCY
                 )
```

BLAS level 3 routine DGEMM (1/2)

DGEMM

Purpose:

```
DGEMM  performs one of the matrix-matrix operations

   C := alpha*op( A )*op( B ) + beta*C,

where  op( X ) is one of

   op( X ) = X   or   op( X ) = X**T,

alpha and beta are scalars, and A, B and C are matrices, with op( A )
an m by k matrix,  op( B )  a  k by n matrix and  C an m by n matrix.
```

BLAS level 3 routine DGEMM (2/2)

- dgemm()

```
subroutine dgemm ( character                       TRANSA,
                   character                       TRANSB,
                   integer                         M,
                   integer                         N,
                   integer                         K,
                   double precision                ALPHA,
                   double precision, dimension(lda,*) A,
                   integer                         LDA,
                   double precision, dimension(ldb,*) B,
                   integer                         LDB,
                   double precision                BETA,
                   double precision, dimension(ldc,*) C,
                   integer                         LDC
                 )
```

LAPACK routine DSYEV

Purpose:

```
DSYEV computes all eigenvalues and, optionally, eigenvectors of a
real symmetric matrix A.
```

- dsyev()

```
subroutine dsyev ( character                          JOBZ,
                   character                          UPLO,
                   integer                            N,
                   double precision, dimension( lda, * )  A,
                   integer                            LDA,
                   double precision, dimension( * )   W,
                   double precision, dimension( * )   WORK,
                   integer                            LWORK,
                   integer                            INFO
                 )
```

LAPACK routine DGETRI (1/2)

Purpose:

```
DGETRI computes the inverse of a matrix using the LU factorization
computed by DGETRF.

This method inverts U and then computes inv(A) by solving the system
inv(A)*L = inv(U) for inv(A).
```

LAPACK routine DGETRI (2/2)

- dgetri()

```
subroutine dgetri ( integer                          N,
                    double precision, dimension( lda, * )  A,
                    integer                          LDA,
                    integer, dimension( * )          IPIV,
                    double precision, dimension( * ) WORK,
                    integer                          LWORK,
                    integer                          INFO
                  )
```

Some other statements

Pointers

- Pointers are variables that instead of storing values store memory addresses
- Inherited from C, where arguments passed by procedures are values instead of memory addresses as in Fortran
- Make faster codes, but they are not always easy to use
- Pointer variables are associated to target variables of the same type by the operator =>
- Pointers are dissociated from targets with the statement NULLIFY(pointer_name)
- The status of a pointer can be checked with the statement ASSOCIATED(pointer_name [, target_name])
- Pointers are very useful for iterative procedures

```
      program pointer_ex1
      use util
c
      implicit none
      integer, parameter :: nrow = 9, ncol = 5
      real(kind=8), target :: matrix(nrow,ncol)
      real(kind=8), pointer :: p_matrix(:,:), p_vec(:)
c
      do j = 1,5
         matrix(:,j) = (/ (real(10*i + j,8), i=1,9) /)
      end do
      write(6,'(/,a,/)') 'Original matrix'
      call wrimat(matrix,nrow,ncol)
c
      p_matrix => matrix(3:5,2:4) ; p_matrix = one
      write(6,'(/,a,/)') 'First modification'
      call wrimat(matrix,nrow,ncol)
c
      p_vec => matrix(8,1:5) ; p_vec = two
      write(6,'(/,a,/)') 'Second modification'
      call wrimat(matrix,nrow,ncol)
c
      stop
      end
```

```
      do j = 1,5
         matrix(:,j) = (/ (real(10*i + j,8), i=1,9) /)
      end do
```

```
Original matrix

     11.0000        12.0000        13.0000        14.0000        15.0000
     21.0000        22.0000        23.0000        24.0000        25.0000
     31.0000        32.0000        33.0000        34.0000        35.0000
     41.0000        42.0000        43.0000        44.0000        45.0000
     51.0000        52.0000        53.0000        54.0000        55.0000
     61.0000        62.0000        63.0000        64.0000        65.0000
     71.0000        72.0000        73.0000        74.0000        75.0000
     81.0000        82.0000        83.0000        84.0000        85.0000
     91.0000        92.0000        93.0000        94.0000        95.0000
```

```
c
      p_matrix => matrix(3:5,2:4) ; p_matrix = one
c
```

```
First modification

     11.0000        12.0000        13.0000        14.0000        15.0000
     21.0000        22.0000        23.0000        24.0000        25.0000
     31.0000        1.00000        1.00000        1.00000        35.0000
     41.0000        1.00000        1.00000        1.00000        45.0000
     51.0000        1.00000        1.00000        1.00000        55.0000
     61.0000        62.0000        63.0000        64.0000        65.0000
     71.0000        72.0000        73.0000        74.0000        75.0000
     81.0000        82.0000        83.0000        84.0000        85.0000
     91.0000        92.0000        93.0000        94.0000        95.0000
```

```
c
      p_vec => matrix(8,1:5) ; p_vec = two
c
```

```
Second modification

     11.0000        12.0000        13.0000        14.0000        15.0000
     21.0000        22.0000        23.0000        24.0000        25.0000
     31.0000        1.00000        1.00000        1.00000        35.0000
     41.0000        1.00000        1.00000        1.00000        45.0000
     51.0000        1.00000        1.00000        1.00000        55.0000
     61.0000        62.0000        63.0000        64.0000        65.0000
     71.0000        72.0000        73.0000        74.0000        75.0000
     2.00000        2.00000        2.00000        2.00000        2.00000
     91.0000        92.0000        93.0000        94.0000        95.0000
```

```
macold_666 % time ./rel_pointer.x

Iter   maxdif
  1   0.38E+01
  2   0.15E+01
  3   0.34E+00
  4   0.62E-01
  5   0.31E-01
  6   0.16E-01
  7   0.78E-02
  8   0.39E-02
  9   0.19E-02
 10   0.97E-03

Maximum original element (index, value, squa
 2257892    8.5999985232    2.9325754079

Maximum difference :   0.17E-02
Average difference :   0.33E-09



real    0m1.333s
user    0m1.206s
sys     0m0.124s
```

```
macold_667 % time ./rel_nopointer.x

Iter   maxdif
  1   0.38E+01
  2   0.15E+01
  3   0.34E+00
  4   0.62E-01
  5   0.31E-01
  6   0.16E-01
  7   0.78E-02
  8   0.39E-02
  9   0.19E-02
 10   0.81E-03

Maximum original element (index, value, square root) :
 7361916    8.5999995236    2.9325755785

Maximum difference :   0.28E-03
Average difference :   0.31E-10



real    0m1.762s
user    0m1.326s
sys     0m0.096s
```

F77 statements

COMMON

- F77 command
- Allows to share information among routines that used it
- Superseded by MODULE

```
      common /name_c/ dpvar1, spvar1,
     &                xvec(6), ivar3
```

DATA

- F77 command
- Allows to initialize variables
- Superseded by F90 variable declarations

```
      data dpvar1, ivar3, xvec
     &     /0.5d0, 4, 6*0.0/
```

Makefiles

Definitions

- A makefile is a file that compiles the code.
- When changes are made to some of the code, only the updated files need to be recompiled.
- The code must be linked again to create the new executable.
- Makefile knows the pieces of a large program that need to be recompiled.
- Makefile has all the sentences to compile and link the code up.
- Such sentences must be prefixed by a TAB character

Important points

- Makefile will recompile those changed from last compilation.
- Comments: must be delimited by hash marks (#).
- Continuation character: backslash (\) the end of the line.
- By default, the first target file appearing in the makefile is the one built.
- The order of the other targets does not matter.
- Makefile is invoked from the command line typing

  make [-f <name_other_than_M(m)akefile] [target]
- In the makefile variables can be defined.
- More useful when preprocessor directives are used

```
      program pointer_test
c
c     Calculate iteratively square root of the elements of a big vector
c
      use module_test
c
      implicit none
c
#if defined (debug)
      integer, parameter :: n = 40
#else
      integer, parameter :: n = 10000000
#endif
c
      real(kind=8) :: xmat(n)
c
#if defined (no_pointers)
      real(kind=8) :: old(n), new(n), tmp(n)
#else
      real(kind=8), target :: old(n), new(n)
      real(kind=8), pointer :: p_old(:), p_new(:), p_tmp(:)
#endif
c
```

```
c
      real(kind=8) :: diffmx, valmx
      real(kind=8), parameter :: thres = 0.001_8
      integer :: indmx, iter
c
      call random_number(xmat)
      xmat = 8.6_8 * xmat
      valmx = maxval(xmat)
      indmx = maxloc(xmat,1)
c
#if !defined (no_pointers)
      p_old => old
      p_new => new
      p_old = 1.0_8
#else
      old = 1.0_8
#endif
c
      write(6,'(/,a,3x,a)') 'Iter','maxdif'
      iter = 0
      do
         iter = iter + 1
c
```

```
c
#if defined (no_pointers)
         new = aprox_sqrt(n,xmat,old)
         diffmx  = maxval(abs(new-old))
#else
         p_new = aprox_sqrt(n,xmat,p_old)
         diffmx  = maxval(abs(p_new-p_old))
#endif
         write(6,'(i3,e11.2)') iter, diffmx
c
         if (diffmx .ge. thres) then
#if defined (no_pointers)
            tmp = old
            old = new
            new = tmp
#else
            p_tmp => p_old
            p_old => p_new
            p_new => p_tmp
#endif
         else
            exit
         end if
      end do
```

```
c
      write(6,'(/,2a,/,i8,2f16.10)') 'Maximum original element ',
     &                         '(index, value, square root) :',
     &                           indmx,xmat(indmx),new(indmx)
c
#if defined(debug)
      call mk_dbg(n,xmat,new)
#else
      call mk_diff(n,xmat,new)
#endif
c
      stop
      end program
```

```
cpp -Dno_pointers pointer_2.F -o rel_nopointer.f
```

```
# 1 "pointer_2.F"
# 1 "<interno>"
# 1 "<línea-de-orden>"
# 1 "pointer_2.F"
      program pointer_test
c
c     Calculate iteratively square root of the elements of a
c
      use module_test
c
      implicit none
c



      integer, parameter :: n = 10000000

c
      real(kind=8) :: xmat(n)
c



      real(kind=8), target :: old(n), new(n)
      real(kind=8), pointer :: p_old(:), p_new(:), p_tmp(:)
```

```
# 1 "pointer_2.F"
# 1 "<interno>"
# 1 "<línea-de-orden>"
# 1 "pointer_2.F"
      program pointer_test
c
c Calculate iteratively square root of the eleme
c
      use module_test
c
      implicit none
c



      integer, parameter :: n = 10000000

c
      real(kind=8) :: xmat(n)
c

      real(kind=8) :: old(n), new(n), tmp(n)
```

```
gfortran -E pointer_2.F  -o rel_pointer.f
```

gfortran -E ≡ cpp

```
      module module_test
c
      implicit none
c
      contains
c
        function aprox_sqrt(n,x,r)
        integer :: n
        real(kind=8), dimension(n) :: aprox_sqrt
        real(kind=8), dimension(:) :: x, r
        real(kind=8) :: two = 2.0_8
        aprox_sqrt = (r + x/r) / two
        return
        end function
c
c
        subroutine mk_diff(ndim,xmat,aprox)
c
c.............
c
        return
        end subroutine
```

$$y = \sqrt{x} \longrightarrow y_{i+1} = \frac{y_i + x/y_i}{2}$$

```
      subroutine mk_dbg(ndim,xmat,aprox)
c
      implicit none
c
      integer :: ndim, i
      real(kind=8), dimension (:) :: xmat, aprox
      real(kind=8), dimension (ndim) :: exact
      real(kind=8) :: diff, maxdif, avedif
c
      exact = sqrt(xmat)
c
      maxdif = maxval(abs(exact-aprox))
      avedif = sum((exact-aprox)**2)
      avedif = sqrt(avedif) / ndim
c
      write(6,'(/,a,e11.2)') 'Maximum difference :',maxdif
      write(6,'(a,e11.2,//)') 'Average difference :',avedif
c
      write(6,'(a3,4a15,/)') '  I','     Original  ',
     &                       '   Approx root ','    Exact root ',
     &                       '    Error      '
      do i = 1, ndim
         write(6,'(i3,3f15.10,e11.2)') i,xmat(i), aprox(i),
     &                        exact(i), abs(aprox(i)-exact(i))
      end do
c
      return
      end subroutine
c
     end module
```

Lines added for debug routine

```
#
# Makefile example
#
FC       = gfortran
#
STD_FLAG = -std=f95 -O2 -c -o
DBG_FLAG = -std=f95 -O0 -c -o
LNK_FLAG = -std=f95 -o
#
LIBS = -I/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include -L/Libr
ary/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib
#
#
def: module_test rel_pointer
	$(FC) $(LNK_FLAG) rel_pointer.x module_test.o rel_pointer.o $(LIBS)
#
rel_no: module_test rel_nopointer
	$(FC) $(LNK_FLAG) rel_nopointer.x module_test.o rel_nopointer.o $(LIBS)
#
dbg: module_test dbg_pointer
	$(FC) $(LNK_FLAG) dbg_pointer.x module_test.o dbg_pointer.o $(LIBS)
#
dbg_no: module_test dbg_nopointer
	$(FC) $(LNK_FLAG) dbg_nopointer.x module_test.o dbg_nopointer.o $(LIBS)
#
```

(TAB)

MAKEFILE EXAMPLE (1/2)

```
#
module_test:
	$(FC) $(STD_FLAG) module_test.o module_test.f
#
#
rel_pointer:
	$(FC) $(STD_FLAG) rel_pointer.o pointer_2.F
#
rel_nopointer:
	$(FC) -Dno_pointers $(STD_FLAG) rel_nopointer.o pointer_2.F
#
dbg_pointer:
	$(FC) -Ddebug $(DBG_FLAG) dbg_pointer.o pointer_2.F
#
dbg_nopointer:
	$(FC) -Ddebug -Dno_pointers $(DBG_FLAG) dbg_nopointer.o pointer_2.F
#
#
clean:
	rm *.o *.mod
```