Introduction to numerical methods in F programming

AN INTRODUCTION TO NUMERICAL METHODS IN F PROGRAMS

The main area of application for F programs is the solution of scientific and engineering/technological problems.

In other words a process which usually involves the solution of mathematical problems by numerical, as opposed to analytical means.

Numerical calculations

The F language was primarily designed to help in the solution of numerical problems, although it is certainly not limited to that purpose.

Consequently, it is extremely important that the writer and the user of such F programs should be aware of the intrinsic limitations of a computer in this area and the steps that may be taken to improve matters.

As it has already been met, a real number is stored in a computer to about six or seven decimal digits of precision with an exponent range of around -10^{38} to $+10^{38}$.

For example $e^{88}=1.651636e38$, but e^{89} can not be calculated because this would require an exponent of 10 (it exceeds the limit that the computer allows.

Any attempt to store a number whose exponent is too large, as here, will create a condition known as **overflow** and will normally cause program fail at this stage of the processing.

Obviously, once a calculation has overflowed, any subsequent calculations using this result will also be incorrect.

A similar situation arises with the number such as

$$e^{-868} = 4.473779e\text{-}38.$$

This situation is known as **underflow**, which is less serious the overflow, since the effect is that the number is too close to zero to be distinguished from zero.

Many computers will report this form of error and store this number as zero.

Or some computer systems do report its occurrence as non-fatal error

Conditioning and stability

A **well-conditioned** problem is one, which is relatively insensitive to changes in the values of its parameters, so that small changes in these parameters only produce small changes in the output.

An **ill-conditioning** problem is one, which is highly sensitive to changes in its parameters, where small changes in its parameters produce large changes in the output.

An example of an ill-conditioned problem is the pair of simultaneous equations,

$$x + y = 10$$
$$1.002x + y = 0$$

The solution is clearly $x = -5000$, $y=5010$.

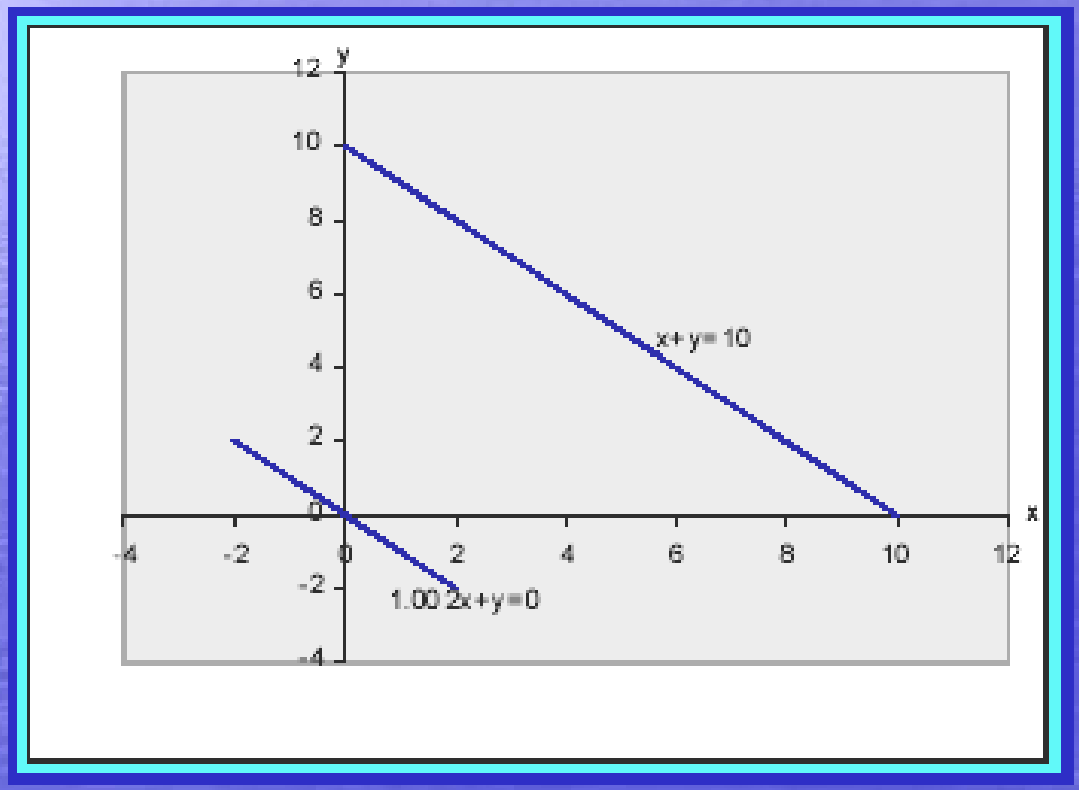
However, if some round-off had led to the second equation being expressed as

$$1.001x + y = 0$$

The solution would have been
$x = -10000$, $y = 10010$

- This is a very great change from the original solution.
- If the round-off error had led the coefficient of x in the second equation to be 1.00 (to four significant digits) then the problem would have been insoluble.

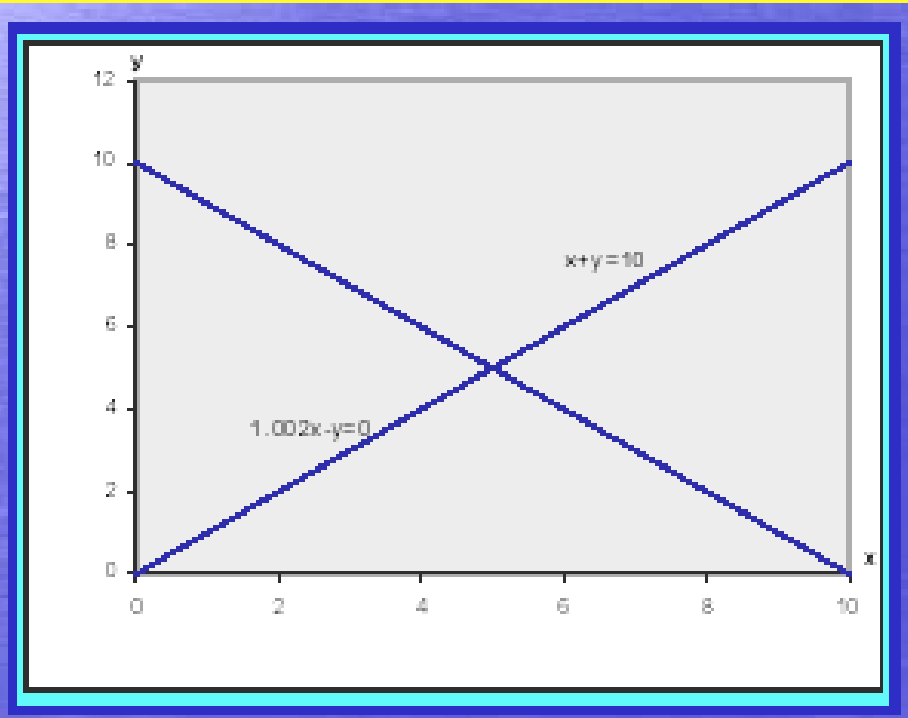
Clearly in this case the reason for this extremely ill-conditioned behaviour is that the two equations represent two straight lines which are almost parallel, and therefore a very small change in the gradient of one will cause a very large movement of their point of intersection.



On the other hand, the two equations

$$x + y = 10$$
$$1.002x - y = 0$$

which have the solution x = 4.995, y=5.002 or x = 5, y=5 respectively. This well-conditioned behaviour is because, in this case the two lines are almost perpendicular to each other.



Data fitting by least squares approximation

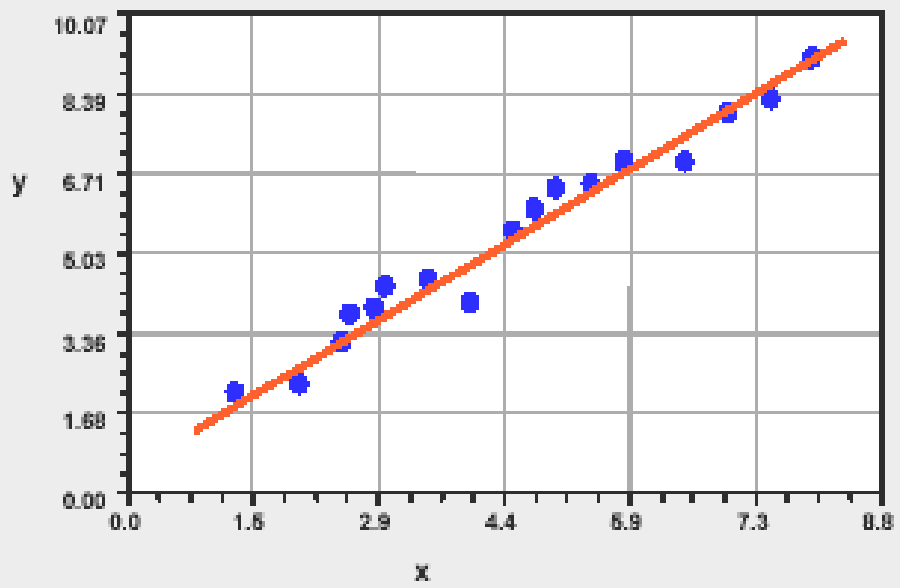
A frequent situation in experimental sciences is that data have been collected which, it is believed, will satisfy a linear relationship of the form

$$y = ax + b$$

However, due to experimental error, the relationship between the data collected at different times will rarely be identical and can typically be represented graphically as shown in the figure drawn below.

- Fitting a straight line through the data in such a way as to obtain the fit, which most closely reflects the true relationship, is, therefore, a widespread need.

- One well-established method is known as the method of least squares.



This method can be applied to any polynomial, or even to more general functions, but for the present but it shall be considered here only the linear case.

The difference between a calculated value and an experimental value y is called residual, and the method of least squares attempts to minimise the sum of the squares of residuals for all the data points.

Some differential calculus, which all are scope of this course, leads to the conclusion that the equation that minimises the square of residuals is when the two coefficients **a** and **b** are defined as follows:

$$a = \frac{\sum x_i \sum y_i - n \sum x_i y_i}{\left(\sum x_i\right)^2 - n \sum x_i^2} \qquad b = \frac{\sum y_i - a \sum x_i}{n}$$

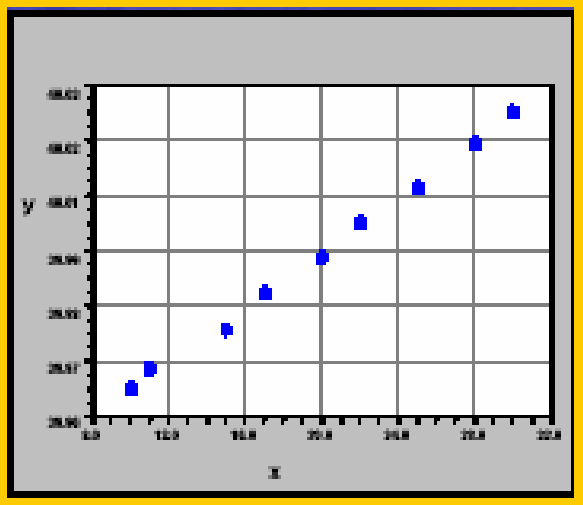
The value of the sum of the squares of the residuals, often referred to as simply the **residual sum**, can be a good guide as to how closely the equation fits the data.

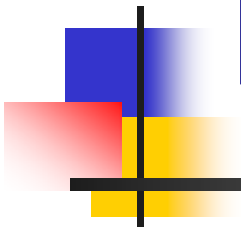
If it is a perfect fit, then all data points will lie on the line and the residual sum will be zero.

Example: Figure drawn below shows the result obtained from an experiment to calculate the linear equation of it.

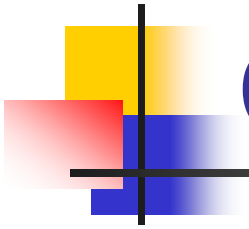
```
subroutine least_squares_line(n,a,b,x,y)
!this subroutine calculates the least squares fit line
!ax + b to x-y data pairs
real, dimension(:), intent(in) :: x, y
integer, intent(in) :: n
real, intent(out) :: a, b
!local variables
real :: sum_x, sum_y, sum_xy, sum_x_sq
!calculate sums
sum_x = sum(x)
sum_y = sum(y)
sum_xy = dot_product(x, y)
sum_x_sq = dot_product(x, x)
!calculate coefficients of least squares fit line
a = (sum_x*sum_y - n*sum_xy) / (sum_x**2 - n*sum_x_sq)
b = (sum_y - a*sum_x) / n
end subroutine least_squares_line
```

```
9
10      39.967
11      39.971
15      39.979
17      39.986
20      39.993
22      40.000
25      40.007
28      40.016
30      40.022
```



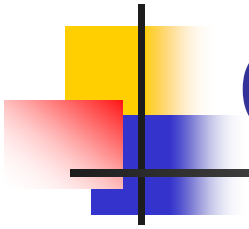


New Numeric Data Type



Complex data type

- The final intrinsic data type is complex which enables the use of complex mathematics
- Complex variables have a real and an imaginary part
- Application areas are mainly electrical engineering



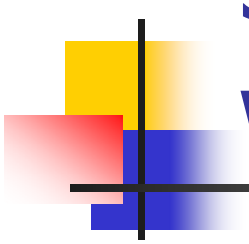
Complex data type

- Complex type data stores two separate real numbers, the first representing the real part and the second representing the imaginary part e.g. (x+iy) =(x,y)

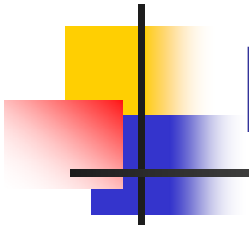
```
complex :: name1, name2
```

A complex constant (1.5, 7.3)

Some intrinsic functions used with complex data type

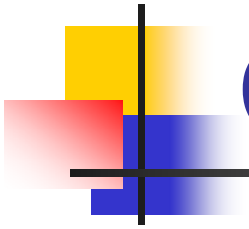


- real(z): z is a complex number, delivers the real part of z
- aimag(z): z is a complex number, delivers the imag. part of z
- cmplx(a): a is real , delivers (a, 0.0)
  a is integer, delivers (real(a), 0.0)
  a is complex, delivers a
- cmplx(x,y): delivers complex value real(x), real(y)



Mixed-mode expressions

- Complex numbers can be combined with real integers in mixed-mode expressions. Real and integers are converted to complex numbers with a zero imaginary part.
- z1=(x1,y1) complex number
- r real number
- r*z1 = (r,0.0)*(x1,y1)

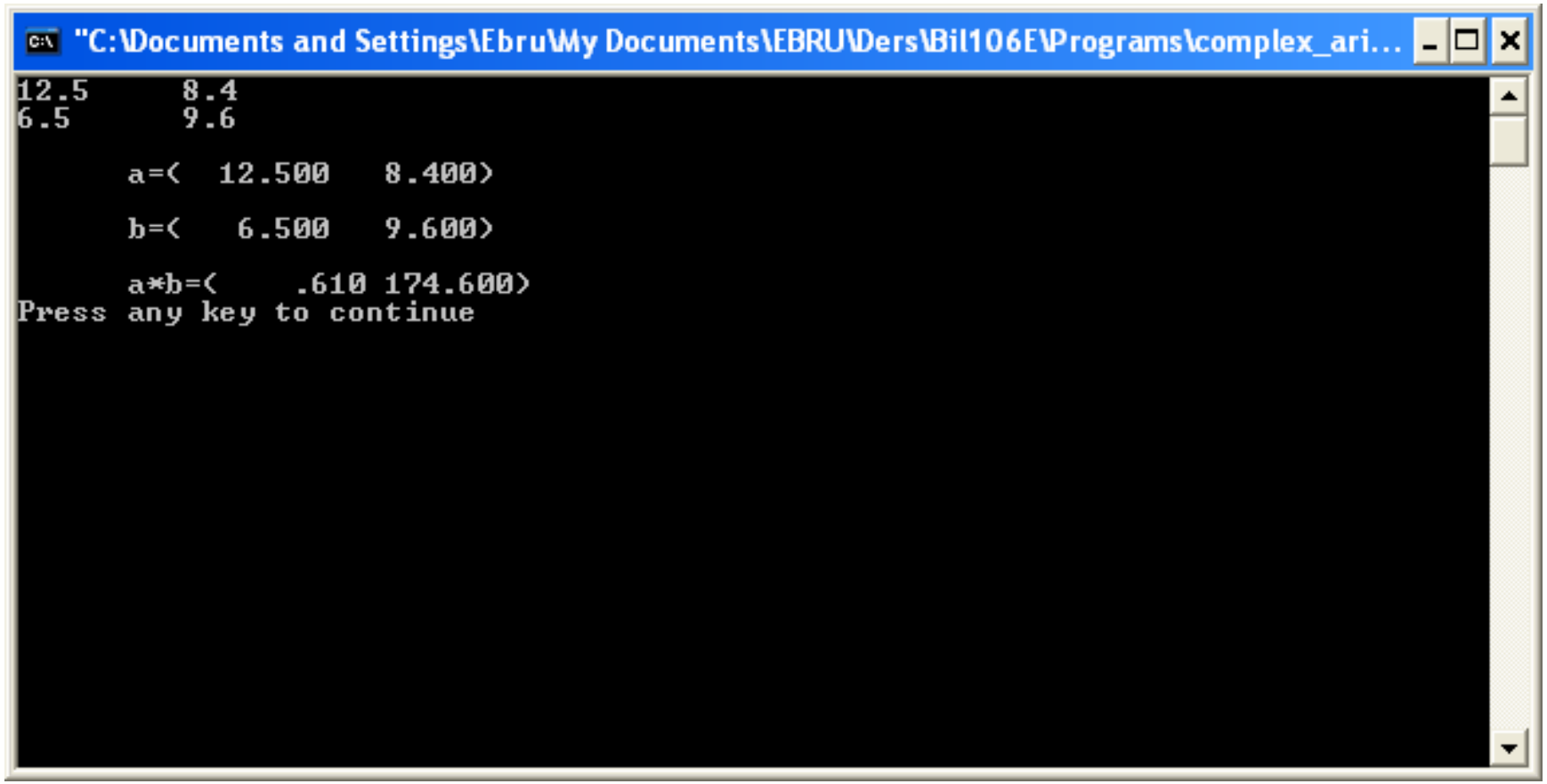


Complex arithmetic example

```
program complex_artihmetic
    complex:: a,b,c

    !Read Two complex numbers
    read "(2f8.3)", a,b
    c=a*b
    !Print data items and their product
    print "(/, t8, a, 2f8.3, a)", "a=(", a, ")"
    print "(/, t8, a, 2f8.3, a)", "b=(", b, ")"
    print "(/, t8, a, 2f8.3, a)", "a*b=(", c, ")"
end program complex_artihmetic
```

Output


"C:\Documents and Settings\Ebru\My Documents\EBRU\Ders\Bil106E\Programs\complex_ari...
12.5 8.4
6.5 9.6
a=(12.500 8.400)
b=(6.500 9.600)
a*b=(.610 174.600)
Press any key to continue

Array processing and matrix manipulation

Matrices and 2-D arrays

- In mathematics, a matrix is a two-dimensional rectangular array of elements (that obeys to **certain rules**!)

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix}$$

Row number

Column number

2D Arrays

F extends the concept of a one-dimensional array in a natural manner, by means of the "**dimension"** attribute.

Thus, to define a two-dimensional array that could hold the elements of the matrix A, we would write :

real, dimension (3, 4 **) : : A**

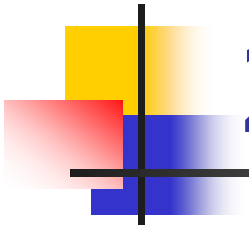
numbers of rows numbers of columns

Matrices and 2-D arrays

- Similarly, a vector is a one-dimensional array of elements (that obeys to **certain rules**!)

$$\begin{bmatrix} A_1 & A_2 & A_3 & A_4 \end{bmatrix}$$

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}$$



2-D arrays

- **`dimension`** attribute:

```
real, dimension(3, 4) :: a
logical, dimension(10,4) :: b, c, d
```

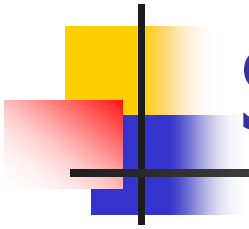
- You can refer to a particular element of a 2-D array by specifying the row and column numbers:

```
a(2,3)
b(9,3)
```

Intrinsic functions

Name	Result
matmul	Matrix product of two matrices, or a matrix and a vector
dot_product	Scalar (dot) product of two vectors
transpose	Transpose of a matrix
maxval	Maximum value of all the elements of an array, or of all the elements along a specified dimension of an array
maxloc	The location in an array where the maximum value first occur
minval	Minimum value of all the elements of an array, or of all the elements along a specified dimension of an array
minloc	The location in an array where the minimum value first occur
product	Product of all the elements of an array, or of all the elements along a specified dimension of an array
sum	Sum of all the elements of an array, or of all the elements along a specified dimension of an array

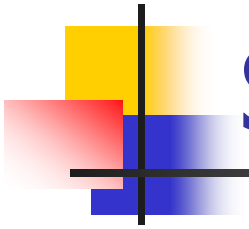
```
program vectors_and_matrices
   integer, dimension(2,3) :: matrix_a
   integer, dimension(3,2) :: matrix_b
   integer, dimension(2,2) :: matrix_ab
   integer, dimension(2) :: vector_c
   integer, dimension(3) :: vector_bc
!  Set initial value for vector_c
   vector_c = (/1, 2/)
!  Set initial value for matrix_a
   matrix_a(1,1) = 1           ! matrix_a is the matrix:
   matrix_a(1,2) = 2
   matrix_a(1,3) = 3           !   [1 2 3]
   matrix_a(2,1) = 2           !   [2 3 4]
   matrix_a(2,2) = 3
   matrix_a(2,3) = 4
!  Set matrix_b as the transpose of matrix_a
   matrix_b = transpose(matrix_a)
! Calculate matrix products
   matrix_ab = matmul(matrix_a, matrix_b)
   vector_bc = matmul(matrix_b, vector_c)
end program vectors_and_matrices
```



Some basic array concepts...

- **Rank**
  number of its dimensions
- Examples:

```
real, dimension(8) :: a                      Rank=1
integer, dimension(3, 100, 5) :: b           Rank=3
logical, dimension(2,5,3,100,5) :: c         Rank=5
```



Some basic array concepts...

- **Extent**
  number of elements in each dimension
- Examples:

```
real, dimension(8) :: a  Extent=8
real, dimension(-4:3) :: b  Extent=8

integer, dimension(3,-50:50,40:44) :: c
integer, dimension(0:2, 101, -1:3) :: d
```

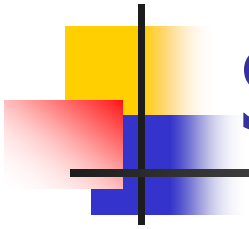
Some basic array concepts...

- **Size**
  total number of elements
- Examples:

```
real, dimension(8) :: a
```
8

```
integer, dimension(3, 100, 5) :: b
```
1500

```
logical, dimension(2,5,4,10,5) :: c
```
2000



Some basic array concepts...

- **Shape**
  - **Rank**
  - **Extent** in each dimension
- Shape of an array is determined by its rank and extent of each dimension
- Example

```
logical, dimension(2,-3:3,4,0:9,5) :: c
shape_c = (/ 2,7,4,10,5 /)
```

Some basic array concepts...

- **Array element order**
  first index of the element specification is varying most rapidly, …
- Example:

```
integer, dimension(4,3) :: a
```

The array element order of arr :

```
a(1,1), a(2,1), a(3,1),a(4,1),
a(1,2), a(2,2), a(3,2),a(4,2),
a(1,3), a(2,3), a(3,3),a(4,3),
```

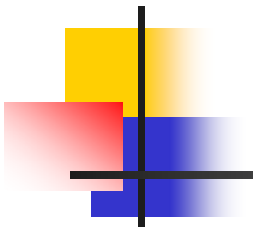
Array constructors for rank-*n* arrays

- An array constructor always creates a rank-one array of values
- The solution to create arrays with rank higher than one you use **`reshape`** function:

```
reshape( (/ (i, i=1,6) /), (/ 2, 3 /) )
```

produces (2 x 3) array such as:

$$\begin{bmatrix} 1.0 & 3.0 & 5.0 \\ 2.0 & 4.0 & 6.0 \end{bmatrix}$$



Example : The implied-do loop given below provides the following matrix

```
integer : :   i,  j
real, dimension ( 2,2 ) : :   a = &
reshape ( (/ (10*i+j, i=1,2 ), j =1,2) /),(/2,2/) )
OR     reshape ( (/ (11, 21, 12, 22) /),(/2,2/) )
```

i=1, j=1 a(1,1)=11
i=2, j=1 a(2,1)=21
i=1, j=2 a(1,2)=12
i=2, j=2 a(2,2)=22

$$\begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$$

Array I/O

Could be handled in three different ways

- As a list of individual array elements:

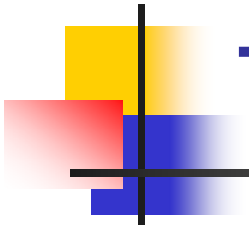
```
a(1,1), a(4,3), …
```

- As the complete array:

```
real, dimension(2,4) :: a
print *, a

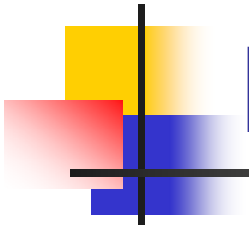
a(1,1), a(2,1), a(1,2), a(2,2), ...
```

- As an array section



The four classes of arrays

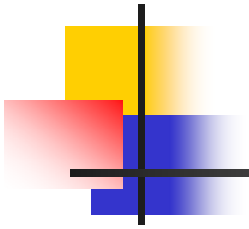
- Explicit-shape arrays
- Assumed-shape arrays
- Allocatable (deferred-shape) arrays
- Automatic arrays



Explicit-shape arrays

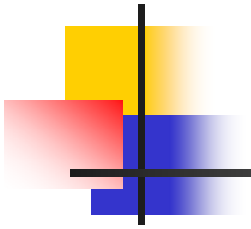
- Arrays whose index bounds for each dimension are specified when the array is declared in a type declaration statement

```
real, dimension(35, 0:26, 3) :: a
```



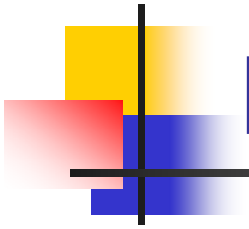
Assumed-shape arrays

- Only assumed shape arrays can be procedure dummy arguments
- Extents of such arrays is defined implicitly when an actual array is associated with an assumed-shape dummy argument
- ```
  subroutine example(a, b)
     integer, dimension(:,:), intent(in) :: a
     integer, dimension(:), intent(out) :: b
  ```
  ```



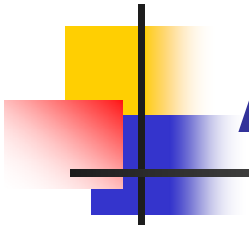
Automatic arrays

- A special type of explicit-shape array
- It can only be declared in a procedure
- It is not a dummy argument
- At least one index bound is not constant
- The space for the elements of an automatic array is created dynamically when the procedure is entered and is removed upon exit from the procedure



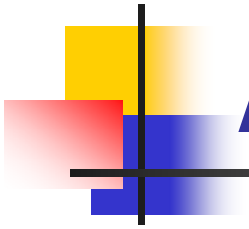
Examples for various type arrays

```
subroutine abc(x)
! Dummy arguments
real, dimension(:), intent(inout):: x ! Assumed shape
! Local variables
real, dimension(size(x)) :: e              !Automatic
real, dimension(size(x), size(x)) :: f     !Automatic
real, dimension(10) :: g                   !Explicit shape
.
.
end subroutine abc
```



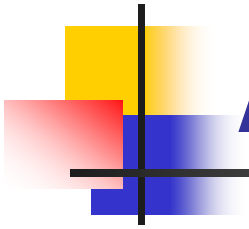
Allocatable arrays

- Allocation and de-allocation of space for their elements is completely under user control
- More flexible: can be defined in main programs, procedures and modules
- Using this type of arrays is slightly more complicated and consists of three steps



Allocatable arrays

- Steps:
  - Array is specified in a type declaration statement
  - Space is dynamically allocated for its elements in a separate allocation statement and the array is used like any other array
  - If the array has been used and no longer required, the space for the elements is de-allocated by a de-allocation statement



Allocatable arrays

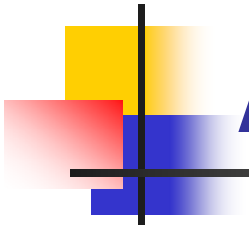
- **Declaration**

```
type, allocatable, dimension(:,:,:) :: allocatable array

real, allocatable, dimension(:,:,:) :: my_array

type(person), allocatable, dimension(:) :: personel_list

integer, allocatable, dimension(:,:) :: big_table
```



Allocatable arrays

`allocate(`*`list_of_array_specs`*`, [stat=`*`status_variable`*`])`

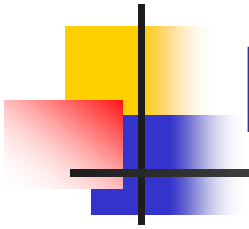
It is recommended to include `stat=`*`status_variable`* element in the statement. It reports the success of the allocation process

Successful `status_variable =0`

Not Successful `status_variable /= 0`

If there is not `stat` element and an error arises during the execution of the allocation statement the program will fail.

```
allocate(big_table(0:50000,1:100000), stat=check)
if (check /= 0) then
   print *, "No space for big_table"
   stop
end if
```

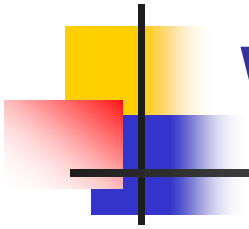


De-allocatable arrays

- An allocated array is de-allocated, making memory space available for other purposes

```
deallocate(list_of_currently_allocated_arrays,[stat=statusvariable])

deallocate(my_array, personel_list, big_table)
```



Whole-array operations

- Whole array operations can be used with conformable arrays of any rank
  - Two arrays are conformable if they have the same shape
  - A scalar, including a constant, is conformable with any array
  - All intrinsic operations are defined between conformable arrays

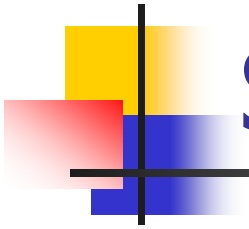
Whole-array operations

- Relational operators follow the same rules:
- Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} -1 & -5 \\ 6 & 2 \end{bmatrix}$$

$$A > B$$

$$\begin{bmatrix} .true. & .true. \\ .false. & .true. \end{bmatrix}$$



Sub-arrays

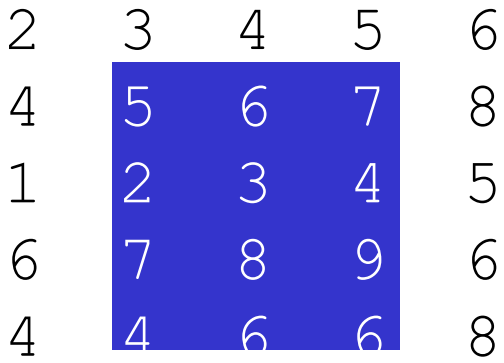
- Array sections can be extracted from a parent array of any rank
- Using subscript triplet notation:

```
first_index, last_index, stride
```

- Resulting array section is itself an array

Sub-arrays

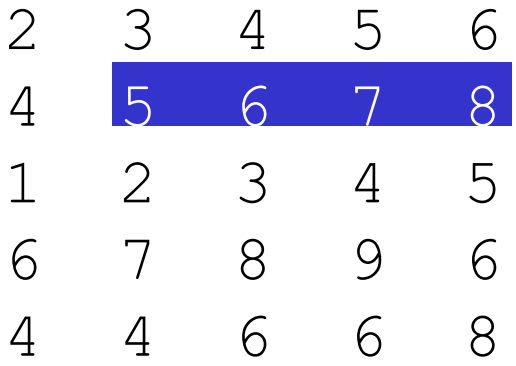
- **`integer, dimension(-2:2, 0:4) :: tablo`**



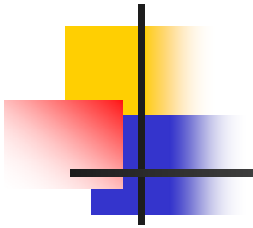
`tablo(-1:2, 1:3)`

Sub-arrays

- **`integer, dimension(-2:2, 0:4) :: tablo`**



`tablo(2, 1:4)`

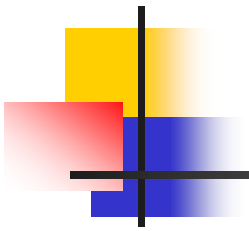


Labwork 1

The results obtained from an experiment to calculate the Young's modulus of the material used to make a piece of wire are given as follows:

Weight	Length
10	39.967
11	39.971
15	39.979
17	39.986
20	39.993
22	40.000
25	40.007
28	40.016
30	40.022

Write a program to calculate the value of Young's modulus by data fitting by least squares approximation

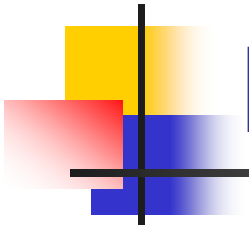


Module

```
      module least_squares
      contains
      subroutine least_squares_line(n,a,b,x,y)
!     this subroutine calculates the least squares fit line
!     ax + b to x-y data pairs
      real, dimension(:), intent(in) :: x, y
      integer, intent(in) :: n
      real, intent(out) :: a, b
!     local variables
      real :: sum_x, sum_y, sum_xy, sum_x_sq
!     calculate sums
      sum_x = sum(x)
      sum_y = sum(y)
      sum_xy = dot_product(x, y)
      sum_x_sq = dot_product(x, x)
!     calculate coefficients of least squares fit line
      a = (sum_x*sum_y - n*sum_xy) / (sum_x**2 - n*sum_x_sq)
      b = (sum_y - a*sum_x) / n
      end subroutine least_squares_line
      end module least_squares
```

Program

```
program least_squares_fit
use least_squares
integer, parameter :: ndim = 100
real, dimension(ndim) :: x,y
real :: a, b
integer :: i, n
open(5,file="least_squares_fit.dat",status="old",action="read")
read(5,*) n
do i = 1 , n
read(5,*) x(i),y(i)
end do
close(5)
call least_squares_line(n,a,b,x,y)
write(6,"(3x,'the input data')")
write(6,"(4x,'i    x              y')")
do i = 1 , n
write(6,"(2x,i3,2x,2(e10.5,2x))") i,x(i),y(i)
end do
write(6,"(/' the equation passing through the data points is')")
if (b > 0.0) then
write(6,"(4x,' y=',e10.5,'*x + ',e10.5)") a,b
else
write(6,"(4x,' y=',e10.5,'*x - ',e10.5)") a,abs(b)
end if
end program least_squares_fit
```

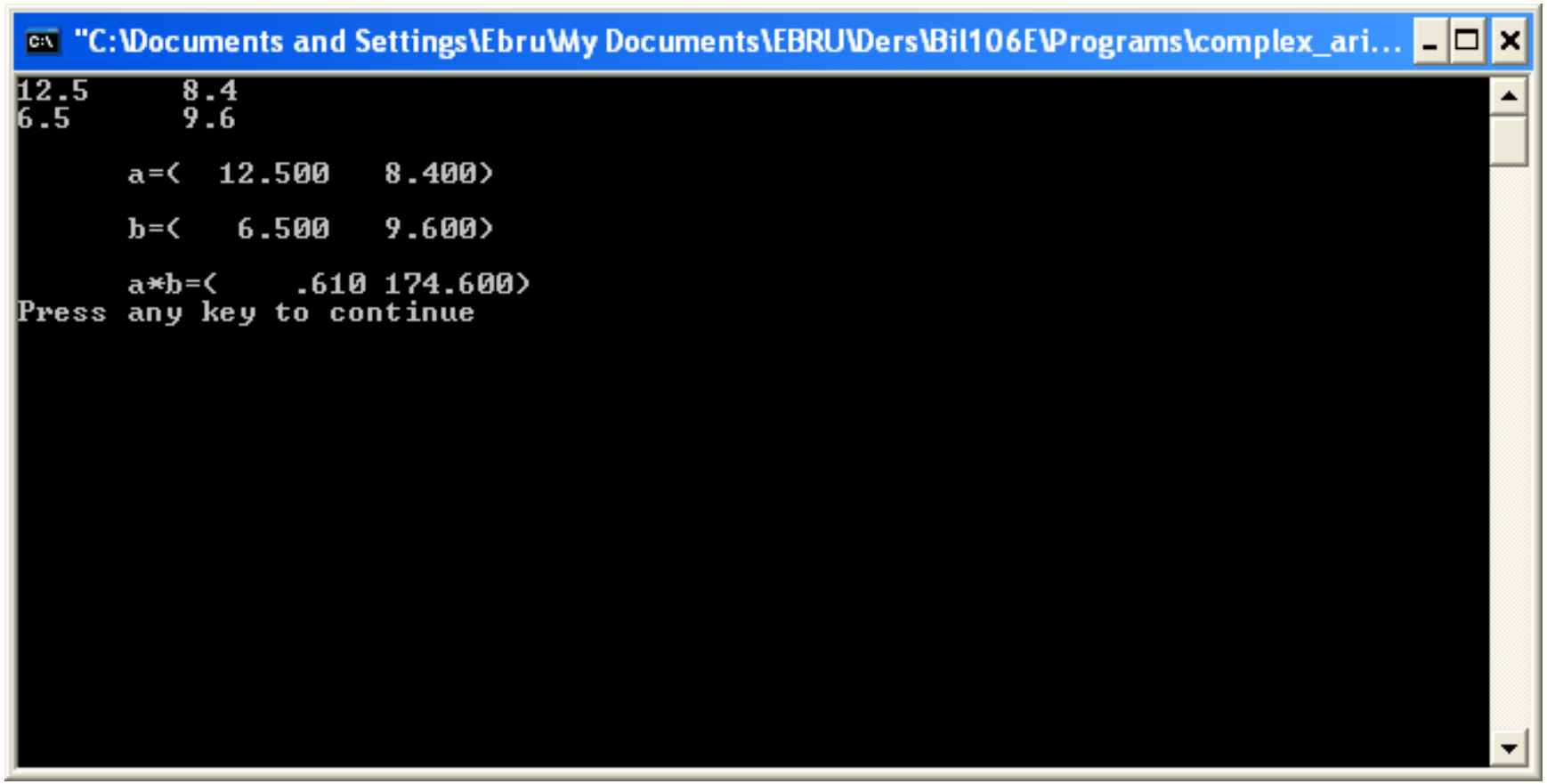


Labwork2

```
program complex_artihmetic
    complex:: a,b,c

    !Read Two complex numbers
    read "(2f8.3)", a,b
    c=a*b
    !Print data items and their product
    print "(/, t8, a, 2f8.3, a)", "a=(", a, ")"
    print "(/, t8, a, 2f8.3, a)", "b=(", b, ")"
    print "(/, t8, a, 2f8.3, a)", "a*b=(", c, ")"
end program complex_artihmetic
```

Output


"C:\Documents and Settings\Ebru\My Documents\EBRU\Ders\Bil106E\Programs\complex_ari...
12.5 8.4
6.5 9.6
a=(12.500 8.400)
b=(6.500 9.600)
a*b=(.610 174.600)
Press any key to continue

Labwork3

```
program vector_manipulation
    integer, dimension(2,3) :: matrix_a
    integer, dimension(3,2) :: matrix_b
    integer, dimension(2,2) :: matrix_ab
    integer, dimension(2) :: vector_c
    integer, dimension(3) :: vector_bc
 !Set initial value for vector_c
    vector_c =    (/1, 2/)
 !Set initial value for matrix_a
    matrix_a(1,1)=1
    matrix_a(1,2)=2
    matrix_a(1,3)=3                ! [ 1   2   3 ]
    matrix_a(2,1)=2                ! [ 2   3   4 ]
    matrix_a(2,2)=3
    matrix_a(2,3)=4
 !Set matrix_b as the transpose of matrix_a
    matrix_b=transpose(matrix_a)           ! [ 1   2 ]
                                           ! [ 2   3 ]
                                           ! [ 3   4 ]
 !Calculate matrix products
   matrix_ab=matmul(matrix_a,matrix_b)     ! [ 14   20 ]
                                           ! [ 20   19 ]
 !Calculate vector_bc
   vector_bc=matmul(matrix_b,vector_c)     ! [ 5   8   11 ]

    print "(/, a, 6i3,a)", " matrix_b  = [", matrix_b, "]"
    print "(/, a, 4i3,a)", " matrix_ab = [", matrix_ab,"]"
    print "(/, a, 3i3,a)", " vector_bc  =[", vector_bc,"]"
end program vector_manipulation
```