Fortran

Mehedi hasan

March 19, 2020

Preface

This is a compilation of lecture notes with some books and my own thoughts. This document is not a holy text. So, if there is a mistake, solve it by your own judgement.

Contents

Listings

Part I

Notes

Chapter 1

Introduction

1.1 History

Fortran (FORmula TRANslation) was the first high-level programming language. It was devised by John Backus in 1953. The first Fortran compiler was produced in 1957. Fortran is highly standardized making it extremely portable (able to run on a wide range of platforms). It has passed through a sequence of international standards those underlined below being the most important.

- Fortran 66 - original ANSI standard (accepted 1972)
- Fortran 77 - ANSI x3.9-1978 - standard programming
- Fortran 95 - ISO/IEC 1539-1 1997 (minor revision)
- Fortran 2003 - ISO/IEC 1539-1 : 2004 (E) object oriented programming interpretability with C
- Fortran 2008 - ISO/IEC 1539-1 : 2010 - Co array (parallel programming)
- Fortran 2018 - Imminent !

Fortran is widely used in high performance computing (HPC). Where its ability to run code in parallel on a large number of process makes it popular for computationability demanding tasks a science and engineering.

1.2 Source Code and Executable Code

In all high-level languages (Fortran, C++, Java, Python, . . .) programmes are written in source code.

This is a human readable set of instructions that can be created or modified on any computer with any text editor. File types identifies the programming language. eg. Fortran file has file types .f90 or .f95.

1.3 Compiler

The job of compiler is to turn source code into machine readable executable code under windows, executable files have file type .exe.

Producing executable code is actually a two-stage process.

- compiling converts each individual source file into object code.
- object code linking combines all the object files with additional library routine to create an executable program.

1.4 Fortran Compiler

1. nagfor
2. IFORT (Intel Fortran Compiler)
3. GNU Fortran (gfortran)
4. Silverfrost FTN95

1.5 Creating and Compiling Fortran Code

You may create, edit, compile and run a Fortran program either

- From a command line
- in an Integrated Development Environment (IDE)

You can create Fortran source code with any text editor. eg. Notepad.

1.6 Some Example Code

1.6.1 Hello World

```
1 program hello
2     implicit none
3     print *, "Hello math 3/1"
4 end program hello
```

Listing 1.1: Hello world in Fortran

1.6.2 Quadratic Equation Solve

The well-known solutions of the quadratic equation $Ax^2 + Bx + C = 0$ are $x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$ The program might look like the following:

```
1 program roots
2     ! Program solves quadratic equations
3     ! a**2+bx+c=0
4     implicit none
5     real a,b,c !Declare Variable
6     real discriminant, root1,root2
7     print*, "Input a,b,c"
8     read*, a,b,c
```

```
 9       discriminant=b**2-4*a*c ! calculate discriminant
10       if (discriminant<0.0) then
11           print*, "No real roots"
12       else
13           root1=(-b+sqrt(discriminant))/(2.0*a)
14           root2=(-b-sqrt(discriminant))/(2.0*a)
15           print*, "Roots are ",root1,root2 ! output
16       end if
17   end program roots
```

Listing 1.2: Quadratic Solver

Chapter 2

Basic Elements of Fortran

2.1 Variable Names

A name is a symbolic link to a location in memory. A variable is a memory location whose value may be changed during execution. Names must:

- have between 1 and 63 alphanumeric character (alphabet, digits and underscore).
- start with a letter.

One should not use Fortran keyword or standard intrinsic (in-built) function as a variable name. Tempting names that should be avoided in this respect include: counts, len, product, range, scale, size, sum, tiny. The following are valid variable names: `SUST_UNITED`, `as_easy_as_123`. The following are not: `Math+Physics` ('+' is not allowed), `999help` (starts with a number), `Hello!` ('!' would be treated a comment not as a part of the variable name)

2.2 Data Types

In Fortran there are 5 intrinsic data types.

1. Integer
2. Real
3. Complex
4. Character
5. Logical

The first three are numeric types while the other two are non-numeric types. It is also possible to have derived data types and pointers.

2.2.1 Integer

Integer constants are whole numbers without a decimal point. e.g. 100, +16, -14, 0, 666. They are stored exactly, but their range is limited; typically -2^{n-1} to $2^{n-1}-1$. Where n is either 16 (for 2-byte integer) or 32 (for 4-byte integer). It is possible to change the default range using the `kind` type parameter.

2.2.2 Real

Real constant has a decimal point and maybe entered as either fixed point, eg. 442.2 or floating point, eg. 4.122e+02. Real constants are stored in exponential form in memory, no matter how they are entered. They are accurate only to a finite machine precision (which again, can be changed using the `kind` type parameter).

2.2.3 Complex

Complex constants consist of paired real number corresponding to real and imaginary parts. eg. (2.0, 3.0) corresponds to $2 + 3i$.

2.2.4 Character

Character constants consists of strings of characters enclosed by a pair of delimiters, which may be either single (') or double (") quotes. eg. `"This is a string", 'Department of Mathematics'`. The delimiter themselves are not part of the string.

2.2.5 Logical

Logical constants may be either true or false.

2.3 A program to compute square root of a number

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \to \sqrt{a}$$

Using `if (...) exit`

```
1  program newton
2      implicit none
3      real a
4      real x, xold
5      real change
6      real, parameter :: tolerence=10e-6
7      print*, "Enter a number:"
8      read*, a
9      x=1.0
10     do
11         xold=x
12         x=0.5*(x+a/x)
13         print*, x
14         change=abs((x-xold)/x)
15         if(change<tolerence) exit
16     end do
17 end program newton
```

Listing 2.1: Newton's method for finding square root (using if)

Using `do while`

```
1  program newton
2      implicit none
3      real a
4      real x, xold
5      real change
6      real, parameter :: tolerence=10e-6
7      print*, "Enter a number:"
8      read*, a
9      x=1.0
10     do while(change>tolerence)
11         xold=x
12         x=0.5*(x+a/x)
13         print*, x
14         change=abs((x-xold)/x)
15     end do
16 end program newton
```

Listing 2.2: Newton's method for finding square root (using do while)

2.4 Declaration of Variable

2.4.1 Type

Variables should be declared (that is, have either data types defined and memory set aside for them) before any executable statements. This is achieved by a type declaration statement of form, eg.

```
integer num
real x
complex z
logical answer
character letter
```

More than one variable can be declared in each statement, eg. `integer i,j,k`

2.4.2 Initialization

If desired variables can be initialized in their type-declaration statement. In this case a double colon (: :) must be used. Thus, the above examples might become

```
integer :: num=2
real :: x=0.5
complex :: z=(0.0, 1.0)
logical :: answer=true
character :: letter='A'
```

Variables can also be initialized with a data statement. eg.
`data, num, x, z, answer, letter / 20,50, (0.09,1.0), .false., 'B'/`
The data statement must be placed before any executable statement.

2.4.3 Attributes

Various attributes may be specified for variables in their type-declaration statement. One such is `parameter`. A variable declaration with this attribute may not have its value changed within the program unit. It is often used to emphasize key physical or mathematical constants. eg. `real, parameter :: gravity=9`

2.4.4 Precision and Kind

By default, `real x` will occupy 4 bytes of computer memory and will be inaccurate in the sixth significant figure. The accuracy can be increased by replacing this type statement by `double precision x` with the floating-point variable now requiring twice as many bytes of memory. Better portability can be used using `kind` parameters. Avoid `double precision` statement by using

```
integer, parameter :: rkind=kind(1.0d0)
```

followed by the declaration for all floating point variable like :

```
real (kind=rkind) x
```

To switch to single precision for all floating-point variable just replace `1.0d0` by `1.0` in the first statement.

Intrinsic functions which allow you to determine the `kind` parameter for different types are

```
selected_char_kind(name)
selected_int_kind(range)
selected_real_kind(precision,range)
```

2.5 Operators and Expression

2.5.1 Numeric Operator

A numeric expression is a formula combining constants, variables and functions using the numeric intrinsic operators given in the following table:

Operator	Meaning	Precedence (1=highest)
**	x^y (Exponential)	1
*	xy (Multiplication)	2
/	$\frac{x}{y}$ (Division)	2
+	$x+y$ (Addition) or $(+x)$ unary plus	3
-	$x-y$ (Subtraction) or $(-x)$ unary minus	3

Repeated exponential is the single exception to the left-to-right rule for equal precedence.

$$a**b**c \rightarrow a^{b^c}$$

2.5.2 Type Coercion

When a binary operator has operands of different type, the weaker type is coerced to the stronger type and the result is of the stronger type. eg. $3/10.0 \rightarrow 3.0/10.0$

2.5.3 Character Operator

There is only one character operator, concatenation, //, eg. `"shah"//"jalal"` gives `"shahjalal"`

2.6 Line Discipline

The usual layout of statement is one pre line. However, there may be more than one statement per line separated by a semi colon; eg, a=1.0;b=1.0; c=100

```
radius = degrees * Pi &
                    / 180.0
```

is same as the single line statement `radius = degrees * Pi / 180.0`

2.7 Remarks

2.7.1 Pi

The constant π appears a lot in mathematical programming. eg, whenever converting between degrees and radians.
If a `real` variable Pi is declared then its value can be set within the program: Pi=3.14159. But it is good to declare it as a `parameter` in its type statement. eg, `real, parameter :: Pi=3.14159`. Alternatively, a popular method to obtain an accurate value is to insert the result

$$\tan(\frac{\pi}{4}) = 1.0$$
$$\Rightarrow Pi = 4.0 * atan(1.0)$$

2.7.2 Exponents

If an exponent ("Power") is coded as an integer it will be worked out by repeated multiplication.
eg, $a**3$ will be worked out as a*a*a
$a**(-3)$ will be worked out as 1/(a*a*a)

For non-integer powers (including whole numbers if a decimal point is used) the result will be worked out by $a^b = (e^{\ln a})^b$
a**3.0 will be worked out something akin to $e^{3.0 \ln a}$. However, the logarithms of negative numbers don't exist. So the following Fortran statement is legitimate:

$$x = (-1) * *2$$

but the next one isn't

$$x = (-1) * *2.0$$

The bottom line is that

- If the exponent is genuinely a whole number, then don't use a decimal point or for small powers, simply write it explicitly as a repeated multiple. eg, a*a*a

- Take special care with odd roots of negative numbers. e.g, $(-1)^{1/3}$; ypu should work out the fractional power of the magnitude, then adjust the sign. eg, write $(-8)^{1/3}$ as $-(8)^{1/3}$

Remember because of the integer arithmetic the Fortran statement x**(1/3) actually evaluates to x**0(=1.0; presumably not intended). To ensure real arithmetic code as x**(1.0/3.0).

A useful intrinsic function for setting sign of an expression as `sign(x,y)` → absolute value of `x` times the sign of `y`.

Chapter 3

Loops

3.1 Types of `do` loop

If a block of code is to be performed repeatedly it is out inside a `do` loop. The basic structure of which is,

```
do
    repeated section
end do
```

Indentation helps to clarify the logical structure of the code. It is easy to see which section is being repeated.

There are two basic types of `do` loops.

(a) Deterministic `do` loops: The number of times the section is repeated is stated explicitly. eg,

```
do i=1,10
    repeated section
end do
```

This will perform the repeated section once for each value of the counter i=1,2,...,10.
The value of i itself may or may not actually be used in the repeated section.

(b) Non-deterministic `do` loops: The number of repetitions is not stated in advance. The enclosed section is repeated until some condition is or is not met. This may be done in two alternative ways. The first requires a logical reason for stopping, whist the second requires a logical reason for continuing looping.

```
do
    repeated section
    if (logical expression) exit
end do
```

or,

```
do while (logical expression)
    repeated section
end do
```

3.2 Deterministic `Do` Loops

The general form of the `do` statement in this case is

```
do variable=value1, value2, value3
    repeated section
end do
```

Note that

- The loop will execute for each value of the variable from value1 to value2 in steps of value3.
- value3 is the stride, it may be negative or positive; if omitted it is assumed to be 1.
- The counter variable must be `integer` type (there could be round off errors if using `real` variables).
- value1, value2 and value3 may be constant (eg, 100) or expressions evaluating to `integer` (eg, 6*(2+j))

3.2.1 Sample program

```
1 program lines
2     implicit none
3     integer n
4     do n=1,100
5         print*, "I must not talk in class."
6     end do
7 end program lines
```

Listing 3.1: A sample program of Deterministic Do loop

Alternatively, the counter (i in the program bellow) may actually be used in the repeated section.

```
1 program doloops
2     implicit none
3     integer i
4     do i = 1, 20
5         print *, i, i*i
6     end do
7 end program doloops
```

Listing 3.2: A sample program of do loop with counter used in repeated section

3.3 Non-deterministic `Do` Loops

The `if (...) exit` form continues until some logical expression evaluates as `.true.`. Then it jumps out of the loop and continues with the code after the loop. In this form a `.true.` result tells you when to stop looping.

This can actually be used to exit from any form of loop.

The `do while(...)` form continues until some logical expression evaluates as `.false.`. Then it stops looping and continues with code after the loop. In this a `.true.` result tells you when to continue looping.

Non-deterministic `do` loops are particularly good for

- summing power series (looping stops when the absolute value of a term is less than some given tolerance)
- Single-point iteration (looping stops when the change is less than a given tolerance)

As an example of the latter consider the following code for solving the Colebrook-White equation for the friction factor λ in flow through a pipe

$$\frac{1}{\sqrt{\lambda}} = -2.0 \log_{10}\left(\frac{k_s}{3.7D} + \frac{2.51}{Re\sqrt{\lambda}}\right)$$

The user inputs values of the relative roughness $\frac{k_s}{D}$ and Reynolds number Re. For simplicity the program actually iterates for

$$x = \frac{1}{\sqrt{\lambda}}:$$
$$x = -2.0 \log_{10}\left(\frac{k_s}{3.7D} + \frac{2.51}{Re}x\right)$$

```
1  program friction
2      implicit none
3      real ksd
4      real re
5      real x ! 1/sqrt(lambda)
6      real xold
7      real, parameter :: tolerence = 1.0e-5
8      print *, "Input Ks/D and Re"
9      read *, ksd, re
10     x = 1.0 ! initial guess
11     xold = x + 1.0 ! anything different from x
12     do while ( abs(x - xold) > tolerence)
13         xold = x
14         x = -2.0*log10(ksd/3.7+2.51*x/re) ! new value
15     end do
16     print *, "Friction factor = ", 1.0/(x*x)
17 end program friction
```

Listing 3.3: Sample program of simple-point iteration

3.4 `Cycle` and `Exit` statement

The `cycle` statement interrupts the current execution cycle of the innermost `do` construct and a new iteration cycle of the `do` construct can begin.

```
do i=1, 10
    A(i)=c+D(i)
    if (D(i)<0) cycle ! if true then the next statement is omitted
    A(i)=1
end do
```

The `exit` statement terminates execution of the innermost (or named) `do` construct.

```
i=0
do
    i=i+1
    if (i>100) exit
    print*, "i is ", i
end do
print*, "Loop finished"
```

3.5 Nested `do` loops

`Do` loops can be nested. Indention is highly recommended here to clarify the loop structure.

```
1  program nestedLoop
2      implicit none
3      integer i,j
4      do i=10, 100, 10
5          do j=1,3
6              print*, "i, j = ", i, j
7          end do
8          print*, "i begins again"
9      end do
10 end program nestedLoop
```

Listing 3.4: A sample program for nested loops

3.6 Implied `Do` loops

This highly-compact syntax is often use to initialize arrays or for I/O or sequential data. The general form is,
`(expression, index = start, end, [stride])`
and like any other `do` loops it may be nested. For example,

```
do i=1, nx
    x=xo + (i - 1) * dx
    print*, x
end do
```

can be condensed to single line,

```
print*, (xo + (i - 1) * dx, i=1,nx)
```

Chapter 4

Conditional Statement / Decision Making

4.1 `if` and `select`

Often a computer is called upon to perform one set of actions if some condition is met and (optionally) some other set if it is not. This branching or conditional action can be achieved by the use of `if` or `case` construct.

4.1.1 The `if` construct

There are several forms of `if` construct

(i) Single statement

```
if (logical expression) ...
```

(ii) Single block of statements

```
if (logical expression) then
    things to be done if true
end if
```

(iii) Alternative actions

```
if (logical expression) then
    things to be done if true
else
    things to be done if false
end if
```

(iv) Several alternatives (there may be several `else if`s and there may or may not be an `else`)

```
1  if (logical expression-1) then
2      things to be done if true
3  else if (logical expression-2) then
4      things to be done if true
5  else
6      ...
7  end if
```

(Here line 5 and 6 are optional)
As with `do` loops, `if` construct can be nested.

4.1.2 The `select` construct

The `select` construct is a convenient (and sometimes more readable and/or efficient) alternative to an `if ... else if ... else` construct. It allows different actions to be performed depending on the set of outcomes (selector) of a particular expression. The general form is,

```
1         select case (expression)
2             case (selector-1)
3                 block-1
4             case (selector-2)
5                 block-2
6             case default
7                 default block
8         end select
```

(Here line 6 and 7 are optional)
expression is an `integer`, `character` or `logical expression`. It is often just a variable.
selector-n is a set of value that expression might take.
block-n is the set of statements to be executed if the expression lies in selector-n.
Case default is used if expression does not lie in any other category. It is optional.
Selectors are lists of non-overlapping `integer` or `character` outcomes separated by commas. Outcomes can be individual values (e.g. 3,4,5,6) or range (e.g. 3:6). The above things are illustrated by the following program.

```
1  program keypress
2      implicit none
3      character letter
4      print *, "Press a key"
5      read *, letter
6      select case (letter)
7      case('A','E','I','O','U','a','e','i','o','u')
8          print *,"Vowel!"
9      case('B':'D','F':'H','J':'N','P':'T','V':'Z','b':'d','f':'h','j':'n',&
10         'p':'t','v':'z')
11         print *,"Consonant!"
12     case('0':'9')
13         print *,"Digit!"
14     case default
15         print *,"Symbol!"
16     end select
17 end program keypress
```

Listing 4.1: Program to find vowel consonant digit or symbol (example of select case)

Chapter 5

Arrays

An array is a group of variables or constants, all of the same type, which are referred to by a single name. The values in the group occupy consecutive locations in the computer memory. An individual value within the array is called an array element; it is identified by tge name of the array together with a subscript pointing to the particular location within the array.

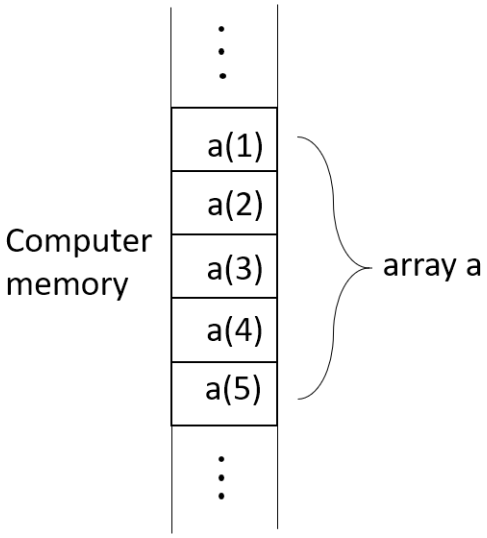

Figure 5.1: The elements of an array occupy successive locations in a computer memory

For example, the first variable shown in figure 5.1 is referred as a(1). The subscript of an array is of type `integer`. Either constants or variables may be used for array subscripts. Arrays can be extremely powerful tools for manipulating data in Fortran. They permit us to apply the same algorithm over and over again to many different data items with simple `do` loops. It is possible to manipulate and perform calculations with individual elements of arrays one by one, with whole arrays at once, or with various subsets of arrays.

Mathematically, we often denote the whole array by its subscripted name. e.g., x for $\{x_i\}$ or a for $\{a_{ij}\}$. Whist subscripted variables are important in any programming languages, it is the ability to refer to ah array as a whole, without subscripts, which makes Fortran particularly valuable in engineering. The ability to refer to just segments of it, e.g., the array section x(4:10) is just like the icing on the cake.

5.1 Some Terminology of Array

5.1.1 `Rank`

The number of subscripts declared for a given array is called the `rank` of the array.

5.1.2 `Dimension`

The `dimension` attribute in the type declaration statement declares the size of the array .

5.1.3 `Size`

The `size` of an array is the total number of element declared in that array.

5.1.4 `Extent`

The number of elements in a given dimension of an array is called the `extent` of the array in that dimension.

5.1.5 Shape

The shape of an array is defined ad the combination of its `rank` and the `extent` of the array in each `dimension`. Thus two arrays have same shape if they have the same `rank` and the same `extent` in each `dimension`.

5.2 Array Declaration

Like any other variable arrays need to be declared at the start of a program unit and memory space assigned to them.

However, unlike scalar variables, array declarations requrire both a type (`integer, real, complex, charact` or derived type) and a `size` or `dimension` (number of elements). For e.g.,

```
real x(100), y(100)
```
or using dimension attribute
```
real, dimension(100) :: x, y
```

We might need to change array size consistently in many places, it is safer practice to declare array size as a single parameter. e.g.,

```
integer, parameter :: maxsize = 100
real x(maxsize), y(maxsize)
```

By default, the first element of an array has subscript 1. It is possible to make the array start from subscript zero (or any other positive or negative integer) by declaring the lower bound as well. For example, to start at zero instead of one,

```
real x(0:99)
```

5.3 Dynamic Arrays

An obvious problem arises, what if the number of points n is greater that the declared size of the array (here, 100)?

One not-very-satisfactory solution is to check for adequate space, prompting the user to recompile if necessary with a larger array size.

```
read (10,*) n
if (n > maxsize) then
    print*, "Sorry, n>maxsize. Please recompile with larger array"
    stop
end if
```

A far better solution is to use dynamic memory allocation; that is, the array size is determined (and computer memory allocated) at run-time, not in advance during compilation. To do this one must use `allocatable` arrays as follows

1. In the array declaration statement, use the `allocatable` attribute, e.g.,
2. ```
   real, allocatable :: x(:), y(:)
   ```

   Note that the shape, but not size is indicated at compile time by a single colon (:).
3. Once the size of the array has been identified at run-time, allocate them the required amount of memory:

   ```
   read (10,*) n
   allocate (x(n), y(n))
   ```
4. When the arrays are no longer needed recover memory by deallocating them:

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

5.4 Elemental Operations

Some times we want to do something to every element of an array.
The expression `x * x` is a new array with elements $\{x_i{}^2\}$.
The expression `sum (x * x)` therefore produces $\sum x_i{}^2$
Using many of these features a shorter version of the above program is given bellow

```
1  ! sample data file
2  !
3  ! ---------------
4  ! | n           |
5  ! | x1   y1     |
6  ! | x2   y2     |
7  ! | ...  ...    |
8  ! | xn   yn     |
9  ! ---------------
10 !
```
```

```
11 program regression
12     implicit none
13     integer n ! number of points
14     integer i ! a counter
15     real, allocatable :: x(:), y(:) ! arrays to hold the points
16     real m,c
17     real xbar, ybar
18     open(10,file="points.dat") ! open the data file; attach to unit 10
19     read(10,*) n
20     allocate(x(n), y(n))
21     read(10,*) (x(i),y(i), i = 1, n)
22     close(10) ! finished with the data file
23     !calculate the best-fit stright line
24     xbar=sum(x)/n
25     ybar=sum(y)/n
26     m=(sum(x*y)/n-xbar*ybar)/(sum(x*x)/n-xbar**2)
27     c=ybar-m*xbar
28     print*, "Slope= ",m
29     print*,"Intercept= ",c
30     print "(3(1x,a10))", "x", "y", "mx+c"
31     print "(3(1x,es10.3))", (x(i), y(i),m*x(i)+c, i= 1, n)
32     deallocate(x,y)
33 end program regression
```

5.5 Matrices and Higher-Dimension Arrays

A m×n arrays of number of the form $\begin{pmatrix} a_1 1 & \dots & a_1 n \\ \vdots & \ddots & \vdots \\ a_m 1 & \dots & a_m n \end{pmatrix}$ is called a matrix (or rank-2 array). The typical element if denoted by $a_i j$. It has two subscripts. Fortran allows matrices and infact, arrays of upto 7 dimension. In Fortran, the declaration and use of a real 3×3 matrix might look like

```
real A(3,3)
A(1,1) = 1.0; A(1,2) = 2.0
A(1,3) = 3.0; A(2,1) = 4.0      etc
```

5.6 Matrix Multiplication

Suppose A, B and C are 3×3 matrices declared by

```
real dimension (3,3) :: A, B, C
```

The statement `C=A*B` does element-by-element multiplication; each element C is the product of the corresponding elements in A and B.

To do "proper" matrix multiplication use the standard `matmul` function:

```
C = matmul(A, B)
```

A similarly use =ful function is that computing the transpose of a matrix :

C = transpose (A)

5.7 Array Constants

Array constants may also be defined. An array constant is an array consisting entirely of constants. It is defined by placing the constant values between special delimiters, called array constructors. The starting delimiter of a Fortran array constructor is `(/` (old) or `[` (modern) and the ending delimiter of an array constructor is `/)` or `]`. e.g., `(/ 1, 2, 3, 4 /)` or `[ 1, 2, 3, 4 ]`

Part II

Assignment

1. Write a program that prompts the user for a positive integer N and output to the screen the partial sum

$$S_N = \frac{1}{2^2} + \frac{2}{3^2} + \frac{3}{4^2} + \cdots + \frac{N}{(N+1)^2}$$

   Find the output of your program for N=20

2. Write a program to evaluate the binomial coefficient

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

   for non-negative integer values of n and r input by the user.
   Use your program to evaluate the binomial coefficients $\binom{6}{4}, \binom{50}{0}, \binom{50}{50}, \binom{40}{4}, \binom{40}{36}$

3. Write a program that will for any positive integer N input by the user, output the Fibonacci sequence 1,1,2,3,5,... up to, but not exceeding N.

Part III

Lab

- Example of `if` statements

```
1  program gradeUsingIf
2      implicit none
3      integer mark
4      character grade
5      do
6          write (*,"('Enter mark (negative to terminate):')", advance="no")
7          read *, mark
8          if ( mark < 0 ) stop
9          if ( mark >= 70 ) then
10             grade = 'A'
11         else if ( mark >= 60 ) then
12             grade = 'B'
13         else if ( mark >= 50 ) then
14             grade = 'C'
15         else if ( mark >= 40 ) then
16             grade = 'D'
17         else
18             grade = 'F'
19         end if
20         print *, "Grade is ", grade
21     end do
22 end program gradeUsingIf
```

Listing 5.1: Mark to Grade convert using if statement

- Example of `select case` statements

```
1  program gradeUsingSelectCase
2      implicit none
3      integer mark
4      character grade
5      do
6          write (*,"('Enter mark (negative to terminate):')", advance="no")
7          read *, mark
8          if ( mark<0 ) stop
9          select case (mark)
10             case(70: )
11                 grade = 'A'
12             case (60:69)
13                 grade = 'B'
14             case (50:59)
15                 grade = 'C'
16             case (40:49)
17                 grade = 'D'
18             case (:39)
19                 grade = 'F'
20         end select
21         print *, "Grade is ", grade
22     end do
23 end program gradeUsingSelectCase
```

Listing 5.2: Mark to Grade convert using select case statement

- Consider the following program to fit a straight line to the set of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ and then print them out with the best-fit straight line. The data file assumed to be of the form shown bellow

```
-------------
| n         |
| x1  y1    |
| x2  y2    |
| ... ...   |
| xn  yn    |
-------------
```

and the best-fit straight line is $y = mx + c$ where,

$$m = \frac{\frac{\sum xy}{n} - \bar{x}\bar{y}}{\frac{\sum x^2}{n} - \bar{x}^2},\ c = \bar{y} - m\bar{x},\ \text{where } \bar{x} = \frac{\sum x}{n},\ \bar{y} = \frac{\sum y}{n}$$

```
1  ! sample data file
2  !
3  ! ---------------
4  ! | n           |
5  ! | x1  y1      |
6  ! | x2  y2      |
7  ! | ...  ...    |
8  ! | xn  yn      |
9  ! ---------------
10 !
11 program regression
12     implicit none
13     integer n ! number of points
14     integer i ! a counter
15     real x(100), y(100) ! arrays to hold the points
16     real sumx, sumy, sumxy, sumxx
17     real m,c
18     real xbar, ybar
19     sumx=0.0; sumy=0.0; sumxy=0.0; sumxx=0.0
20     open(10,file="points.dat") ! open the data file; attach to unit 10
21     read(10,*) n
22     ! read the rest of the points, one  per  line and add to sums
23     do i=1,n
24          read(10,*) x(i),y(i)
25          sumx=sumx+x(i)
26          sumy=sumy+y(i)
27          sumxy=sumxy+x(i)*y(i)
28          sumxx=sumxx+x(i)*x(i)
```

```
29      end do
30      close(10) ! finished with the data file
31      !calculate the best-fit stright line
32      xbar=sumx/n
33      ybar=sumy/n
34      m=(sumxy/n-xbar*ybar)/(sumxx/n-xbar**2)
35      c=ybar-m*xbar
36      print*, "Slope= ",m
37      print*,"Intercept= ",c
38      print "(3(1x,a10))", "x", "y", "mx+c"
39      do i=1,n
40          print "(3(1x,es10.3))", x(i), y(i),m*x(i)+c
41      end do
42  end program regression
```

Listing 5.3: Example of array using best-fit line problem

- l
- l

Part IV

Questions from Past Years

5.8 2019

1. (a) (2 marks) Briefly explain various types of constants used in FORTRAN.

   (b) (4 marks) What do you mean by variable in FORTRAN? Write the rules for naming a variable in FORTRAN. Explain which of the variables names in FORTRAN are valid or invalid and why?
   i) Mathematics **ii)** STOP **iii)** ORPS **iv)** width

   (c) (2 marks) How is the type of a variable in FORTRAN specified? Explain with example.

   (d) (4 marks) Write the FORTRAN expressions for each of the following mathematical expressions:

   i. $\left|\sqrt{a-b^2}-\frac{c^2}{a/b}\right|$
   ii. $\cos(2x-y)+|x^2+y^2|+e^{xy}$
   iii. $\cos^{-1}x+\tan x^2+e^{mx}+x^3y^2e^{|x|}$
   iv. $(a^n)^m+a^na^m+\ln(a+bx^2)$

2. (a) (2 marks) The following IF construct is incorrect, why? Also write the correct form with flow chart.

```
IF  (x<y) c=a+b*d
    ELSE IF  (x==y)  c=d
    a=b
    ELSE IF  (x>y)  THEN  c=a-b/e+f
    ELSE  e==f+g
          ENDIF
```

   (b) (6 marks) Write the appropriate subprogram and main program to
   i) interchange two values **ii)** sort an ascending array A(1),A(2),...,A(N) to a descending array using the subprogram in (i)

   (c) (4 marks) THe commission on a clerk's total SALES is as follows:

   i. IF SALES¡$50, then there is no commission
   ii. IF $50≤ SALES ≤ $250, then commission =10% of SALES
   iii. IF SALES¿$250, then commission =$20+8% of SALES above $250

3. (a) (4 marks) What do you mean by a DO loop? Write the general form of DO statement and explain its functionality. What is the terminal statement of a DO loop?

   (b) (2 marks) Find the final value of K after the following FORTRAN program segment is executed:

```
      K=2
10  DO  20  I+3,8,2
IF  (1  .EQ.  5)  GO TO  20
      K=K+1
   20  CONTINUE
      k=2*k
```

(c) (6 marks) Write an algorithm, draw flow chart and write the FORTRAN program that reads an integer N and prints the sum of the series: $1^3 + 2^3 + 3^3 + \cdots + N^3$

4. (a) (6 marks) Write a FORTRAN program to sum the following series using FOR loop and WHILE loop severally $(1 + \frac{1}{2})(1 + \frac{1}{2} + \frac{1}{3}) \ldots (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N})$

(b) (4 marks) If M=12345, N=56789, A=456.123 and B=678.999, find the output after the following statements:

i.

```
print 10 M,N,A,B
10 format (3x,18x2x17,2x,x(F8.3,2x))

print 20 A,B,M,N
20 format ('1', E12.2||1x,E12.2||1x,218)
```

(ii) (2 marks) Locate errors (if any) and correct them:

i.

```
read (5,10) A,K,M,Z
10 format (F8.0,2x,2318)

write (6,12) A,B,M
12 format (2x,F8.2,2x,I8,x,F8.3)
```

5. (a) (5 marks) Define an array, dimension statement and parameter statement in FORTRAN. Find the number of elements in the array: DIMENSION P(1:10), T(2:5,1:4),S(0:3,2:5,3:4)

(b) (2 marks) Which of the following array names in FORTRAN are valid or invalid and why? **i)** xx(3*4,6); **ii)** XY(N+3,M); **iii)** YZ(A,L,P); **iv)** ZX(1=1+1)

(c) (5 marks) Four test are given to a class of 10 students. Write a program that calculates the average score of each student and the average score in each test.

6. (a) (3 marks) What do you mean by main program and subprogram? What are the main differences between a function subprogram and a subroutine subprogram?

(b) (5 marks) What is meant by format directed input and output? Explain the I-format, F-format, E-format and A-format specification statement in FORTRAN.

(c) (5 marks) Briefly explain the file processing in FORTRAN. Explain open fine and close file in FORTRAN.