Basics of Fortran (77)

S. C. Phatak *

Institute of Physics

Bhubaneswar 751 005

December 26, 2002

1 Preliminaries

Fortran (Formula Translation) is a very old computer language. It has undergone several evolutionary stages and at present fortran 95 is the standard fortran. The previous version (fortran 77) is also common among older people. It is believed that the demand for fortran is going down in favor of C++. Particularly, big laboratories (CERN for one) has decided to switch to C++. As a result, it is possible that fortran will be superseded by C++. But, many times in the past, people have announced that the fortran will die within a short time and it has always bounced back. Over the time it has imported concepts from other computer languages and assimilated them. So it is possible that fortran will still survive. Personally, I would prefer that it survives so that my investment will be intact and I will not have to learn new

*email : phatak@iopb.res.in

computer languages. For you, you have a choice of either learning fortran or C++. If I were you, I would learn both but would choose C++ for writing programs.

The Fortran, as the name suggests, is best suited for numerical computations. That is why it is popular among physicists, chemists, mathematicians and engineers. The handling of alphanumeric strings is not so good in fortran but that is not such a handicap because one doesn't need this aspect very much while doing numerical computations. Like any language FORTRAN has its syntax and that is what we shall be learning here. We shall be following a utilitarian approach so we shall not dwell much on the structure of fortran per se but we shall learn fortran by using it. So, in the following, I shall introduce some concepts and immediately give examples to illustrate the concept. Some of the examples will be solved examples and many more will have to be solved by you.

The way the fortran programs are used on computers is as follows. First one makes (writes) a fortran program on computer. That is, one creates a file containing a collection of fortran statements. Invariably a fortran program has .f or .f77 as an ending character string (extension). The computer cannot understand this fortran program as such and it has to be converted into a program that the computer understands. This is done by a fortran compiler and the process is called compilation of the program. The translated program is in machine language and often called as a.out. But it can have any name. Before the compiler creates a machine language program, it creates an intermediate file which has an ending .o. This file is called the object code. So, let us assume that you have made a program called *foo.f* (Here foo is a generic name of a file. That is, foo can be replaced by any character string). On compilation, an object file *foo.o* and an executable a.out is created by the computer. The command for doing this is

f77 foo.f

Other possible names of the compiler are fort, g77, f90, lf90 (only in the Institute of Physics and on some of the machines) etc. The output of the command is foo.o and a.out. To run the program, type

a.out

or

./a.out

at the command prompt. The program will then be executed.

The steps involved in computing using fortran

1.	make a program file (eg *foo.f*) using any editor of your choice
2.	compile the file and make an object code (*foo.o*) and an executable file (usually *a.out*)
3.	execute the program (*a.out*)
4.	check the results for errors, correct the errors in the fortran program and go back to step 2

The Fortran compiler, by default, creates executable file named a.out after compilation. The problem with default is every compiled file is a.out so one doesn't know what it does if you have a number of programs. Also, every time you compile a new program, old a.out is erased and new a.out is written in it's place. So, it is convinient

to follow a convention where the executable of a Fortran program has a name similar to the name of the fortran program. The convention I follow is to name the executable as foo. The command to generate this executable is

f77 -o foo foo.f

It appears that the object file foo.o is of no use. As mentioned above, the object file is a program which is between a fortran program and its executable code. It is useful when one has a number of subunits (subroutines, functions, libraries) in a fortran program and if one is modifying only a part of a program, it doesn't make sense to compile all the other program units again and again. In this case, one can compile the program that is changed and use link the object code thus created with the object codes of other program units. This saves a lot of compilation time, particularly if one has tens of other program units. During the course of these lectures we will not be dealing with complicated and lengthy programs and as a result we will not be using the object files in the course.

2 format of Fortran Programs

The fortran programs follow a certain format. We shall first consider the format of fortran 77 statements. In fortran 77, the statements begin in column 7 of a line and should not stretch beyond 72nd column. The first column is reserved for declaring comment statements. If one types c in the first column, that line is considered as a comment. Columns 2 to 5 are reserved for labeling (numbering) the line. If the columns 7-72 are not sufficient to complete a statement, it may be continued on the next line by typing any character in the sixth column of the line. There is a provision for having sufficiently large numbers of successive continuation statements, although

it is not a good practice to have very lengthy statements. For the clarity of the code, one should break up long expressions into small ones by defining intermediate variables. Fortran 77 does not distinguish between upper case and lower case letters. This format of programs in fortran 77 is a relic from the days when the computer codes were punched on cards and fed into card reader for communication with the computer.

The structure of fortran statements

	col. no.	
1.	1	comment declaration
2.	2-5	label (number) of a statement
	& 3.	6 marker for continuation of a statement
4.	7-72	fortran statement proper
5.	73	not interpreted by the compiler

Many of the restrictions of computer program format in fortran 77 have been removed in fortran 90. For example, a line in fortran 90 may contain unto 132 characters (columns) and may include more than one statement, each statement being separated by ;. The comment is declared by ! with everything following ! being

treated as a comment statement. A statement may start from first column and for continuing the statement on next line, the last character on the line (before any !) must be &. The statement label (number) should be the leading characters in a new line and it must be separated from the contents of the line by a blank. Clearly, fortran 90 is more suited for computers having terminals, which is a common practice now.

Computer codes in fortran generally consist of a number of program units called programs, subroutines or functions[1]. There is only one program, usually called main program and the execution of the code starts from this program. The code may not have more than one main program. The execution of the code starts from the begining of the main program and continues one line at a time. The main program may 'call' (transfer the execution) to subroutines or functions. On completion of the execution in the subroutine or function the control is returned to the line where the the program unit calls the function to the subroutine and continue from there onwards. Thus the role of the function or subroutine is to modularize the code. In principle, one could replace the calling line by the code in the function or subroutine.

Each program unit (main program, subroutine or function) is arranged in the order given below.

1. Declaration of the unit: This is the first statement of the unit. the statements are program foo [for main program], function foo(foo1,foo2) [for functions] or subroutine foo(foo1,foo2,foo3) [for subroutines]. The quantities in the bracket following the function and subroutine name are the objects that are transfered from the calling program unit to the function or subroutine or vice versa. These should be identical in the calling program and function or subroutine.

[1]The details of subroutines and functions, how they are defined and used, etc is discussed later

2. Declaration statements: These statements declare the variables or constants defined and used in the program unit. These statements also include the declaration of common blocks (which include the variables used in different program units), data statements which define the values of constants or variables etc.

3. Function declarations: These statements define in-line functions as short cut. For example, one may be calculating the relativistic energy from the momentum and mass of a particle at several places in the program. Instead of defining the expression at each place, one may define it here in terms of two variables (say a and b) and use this definition in the program.

4. Executable statements: This is the main body of the program unit.

5. stop (for main program) or return (for subroutine or function) statement. This signals that the task of the program unit is over and the control may be transfered to the calling program unit or the program may be terminated (for the main program). The stop or return statements may occur anywhere in the program and may appear more than once.

6. end. This is the last statement of a program unit. It signifies the end of the program unit so whatever follows must be another program unit.

3 Variables and Constants

The objects used in fortran are of several types. There are two kinds of objects, variables and constants. As the name suggests, constants are expected to be constants, not changing during the execution of the program. The (values of the)variables are

the objects which may change during the execution of the program. Examples of the constants are the values of π, e, $\sqrt{2}$ etc. You will need to call these constants as well as the variables by some name. We shall see below how one can declare the constants and variables.

As in case of mathematics, the objects in fortran are of different types: integers, real numbers and complex numbers. In addition, one can have logical objects and objects representing characters or character strings. Primary function of a fortran program is to do numerical computations so mostly one requires the first three types of objects. Logical objects are required when one is comparing two objects or when one is using logical operations to decide on options for computing. The character strings are required primarily when one is opening files for reading or writing. Then the file names are to be supplied as character strings. Fortran is weak in operations on character strings, but then, one rarely needs character strings during numerical computations.

3.1 Integers

As you know, the set of integers is infinite but countable. In computer, however, there are limitations on how large an integer one can store. The smallest unit of the memory that is accessed in fortran is one byte, which consists of eight bits. This means that a byte can represent any integer between 0 and $2^8 - 1 = 255$ If one wants to include negative integers as well, one of the bits has to be used for the sign and as a result, numbers between -128 and 127 can be represented in one byte. One may decide to use more than one byte to represent an integer. In that case, you will be able to represent larger integers. But no matter what you do, there is a limit on the number of integers one can represent in a computer. Normally, the fortran compilers

support two kinds of integers. One, called integer, is represented by four bytes. The other is called long integer which is represented by eight bytes. You can work out the largest integers that can be represented in these two systems.

It is very important to realize this limitation and monitor the effects of this limitation on the computed results. One of the consequences of this limitation is, if one attempts to represent an integer larger than the largest allowed value, the number stored in the computer is some other number. Actually, beyond the largest number, the numbers start from smallest (negative) number. This is called integer overflow. The computer does not warn you about it so you need to be careful about this and watch out for such overflows and handle them appropriately, else it may result in the output which is wrong. Below, we shall write a fortran program to determine when the integer overflow occurs.

3.2 Real Numbers

In real world, the set of real numbers also consist of infinite elements but they are not countable. That is, there are infinity of real numbers between any two real numbers. Again, because of finite storage space available, one cannot represent all real numbers in a computer. The difficulties involved in case of real numbers are two-fold. For on thing, there is an upper limit on the largest real number one can store. Besides this, the real numbers represented on a computer have granularity. That is, there are not infinity of numbers (represented in a computer) between any two real numbers. In fact, one can find two nearest numbers on a computer such that there are no computer-represented numbers between them.

As in the case of integers, there are two representations of real numbers. One which uses four bytes (single precision) and one which uses eight bytes (double

precision). Some compilers allow sixteen bytes (quadrupole precision) but I don't recommend its use, partly because then the computer code is not transportable. Also, the quadrupole precision computations are done using compiler software and therefore very slow.

The real numbers are represented in the computer in an exponential format. That is a number is represented as $\pm 0.nnnn \times 10^{\pm mmmm}$. So, out of the allocated space of a real number, two bits are used up to define the two signs and rest are divided between the fraction (the mantissa) and the exponent. This means that there is a limit on the largest *as well as* smallest number (apart from sign) that can be represented in a computer. Further, because there are finite number of bits allocated to the mantissa, the number system in the computer is granular. This also means that, not only the operations on real numbers may result in overflows, it may result in underflows, In case of underflows, the number is replaced by zero. In many situations this is harmless. But, in case you are dividing this number, you get division by zero, leading to overflows. So, as in the case of integers, one needs to watch out for the underflows and overflows of real numbers. And as in the case of integers, one needs to take care of these.

3.3 Complex Numbers

Complex number is essentially a pair of real numbers subjected to the complex arithmetic operations. So, as in case of the real numbers, the complex numbers represented in the computer are restricted by the smallest and largest numbers and are granular. Most of the compilers allow the definition of single precision complex numbers. That is the real and imaginary parts of a complex numbers are single precision real numbers. Some compilers may permit double precision complex numbers. In case the

compiler available to you does not permit double precision complex numbers and you need better accuracy, you must do the complex number operations using your own functions.

3.4 Logical quantities

Logical quantities have two values, yes and no. So, one needs one bit to define a logical quantity. But, a byte being the smallest unit of memory that is accessed by fortran, logical quantities are of length one byte. Logical constants are yes or no (or true or false).

3.5 characters

Character type includes the characters available on the keyboard. That is all upper case and lower case letters, numerals and other characters normally used. One usually needs strings of characters to define file names. The string of characters can be defined as an array of characters. We will not need to learn the manipulations of strings (cutting strings, concatenating strings etc). If you really have to do it, fortran is not the language you should be using. It is best to use C or C++ for that.

4 Declaration of Objects

In a fortran program one needs to give names to the quantities for future reference in the computer program. For this one needs to declare the type of quantity one is defining. There are certain conventions one must follow in defining the objects. The names of the objects must begin with a letter (upper case or lower case). The

name may be as long as 31 characters (but you will hardly ever define a name longer than seven or eight characters) and may not include the characters . , / * <> - + = ( ) ! # $ % & { } [and]. Fortran 77 does not distinguish between upper case and lower case letters. Fortran 90 can but there is a compiler switch which allows one to have fortran 77 functionality. So, one can never run out of names. **A good programming practice is to define the names which have some connection to the variable being defined. For example, it is always a good idea to define π with a constant name pi instead of using some name 'a' or some such thing.**

	statement	meaning
1.	implicit none	Every variable and constant must be declared
2.	implicit real (a-c,p-y)	Every variable and constant begining with letters a-c and p-y are real (single precision) the word real here may be replaced by integer, real*8, complex*8 etc, with obvious meaning
3.	integer i, j, a(20), b(20,20)	variables (or constants) i, j, a and b are integers with a being a matrix of rank 1 and dimension 20 and b a matrix of rank two and dimension 20×20. the word integer here may be replaced by real, real*8, complex*8 etc, with obvious meaning
4.	common /foo1/a1,a2	The objects (defined else where) are common to another program unit. These objects may be of any type but they must have same meaning in the program units where the common foo is declared. The name of the common is foo1
5.	data i /2/	Integer variable i is assigned value 2 This is a handy way of defining long arrays of variables

In fortran 90, every constant or variable has to be declared. The compiler does not allow the use of variables which are not declared. That is nice because it avoids mistakes of using quantities without defining and declaring them. In fortran 77, the default option is the variables need not be declared. The variables and constants begining with i, j, k, l, m and n are integers and all the others are real numbers (single precision) by default. This saves a lot of typing but could lead to errors

which are very difficult to find out. Of course it doesn't happen in small programs but imagine using an undeclared variable on 230th line of the program. Fortunately, there is an option of demanding declaration of all variables in fortran 77 and I will recommend that this practice is followed. .

Declaration of objects is done in the begining of a program unit. I list the declaration statements used in fortran 77 in the table here.

5 Operators

Operators operate on the objects to produce new objects. Operators are unary (operating on one object) or binary (operating on two objects). The syntax for using an operator is usually:
object (operator) object
and the result is another object. For unary operators, the first object is absent. The types of operators in fortran are arithmetic, relational, logical and string operators. The new object defined by means of an operator is usually assigned to a variable and the expression is of the form
obj1 = obj2 (operator) obj3
Note that the 'equality' sign does not mean equal to but assign to. That is, assign the result of obj2 (operator) obj3 to obj1. Thus, a statement
obj2 = obj2 (operator) obj3
is a perfectly acceptable statement in fortran although it is nonsense arithmetically. One can combine the result of an operation with another operator and define complex statements. In that case, the order of operation of the operators is decided by the precedence rule defined below.

5.1 arithmetic operators

Arithmetic operators consist of addition (+), subtraction (-), multiplication (*), division (/) and 'raised to power' (**). These operators are binary but addition and subtraction may also be unary operators. The precedence rule for the arithmetic operators is

1. The evaluation of complex expressions is done from left to right.

2. Expressions within brackets are evaluated first.

3. Among the arithmetic operators exponentiation is done first, that is followed by multiplication and division and finally addition and subtraction is done in the end.

General rule of thumb is to use the brackets liberally so that you are sure about what the computer is doing since the expressions in the brackets are evaluated before anything is done. Most of the fortran codes you will be writing will consist of arithmetic operators operating on arithmetic objects and therefore it is very important to have a good understanding of the precedence rules given above.

Most of the time one would be operating the operators on objects of same type. Then the resulting object is also of the same type as the objects on which the operator operates. It is possible to mix the objects in fortran 77 as well as fortran 90. In that case one must know the object type of the result. The rule of thumb is, if an operator operates on two objects of different type, the resulting object type is more complex among the two. This is illustrated in the accompanying table.

object1	object2	result
integer	integer	integer
integer	real	real
integer	complex	complex
real	integer	real
real	real	real
real	complex	complex
complex	integer	complex
complex	real	complex
complex	complex	complex

There is one important property of division of two integers which is peculiar and can be of use in writing programs. By the rules defined in the table, division of two integers is defined to be an integer in the computer arithmetic. But this is generally not an integer but a real number. Actually, in arithmetic computations in a computer, the division is assigned the integer value of the result. Thus, division of 7 and 2 is assigned a value 3 and not 3.5. One can use this property of division to test whether a number is divisible by an integer or not. For example, to check whether an integer n is even or odd one can compute (n/2)*2 and compare it with n itself. The result is true (the two quantities are equal) if n is even and false otherwise. This happens because n/2 is computed first and that result is n/2 if n is even and (n-1)/2 if n is odd.

5.2 Relational operators

The relational operators compare two arithmetic objects. Obviously, the two should be of same type. In case the objects that are compared are not of same type, the less complex object is converted to a more complex object before comparison. So, when comparing an integer or a real number with a complex number, the integer or the real number is converted to a complex number. Further, a meaningful comparison can only b done with scalar objects. The relational operators in fortran 77 and fortran 90 and their meaning is shown in the accompanying table.

	fortran 77	fortran 90	
1.	.lt.	<	less than
2.	.le.	<=	less than or equal to
3.	.eq.	==	equal to
4.	.ge.	>=	greater than or equal to
5.	.gt.	>	greater than
6.	.ne.	/=	not equal to

The relational operators are binary, so the operator is in between two objects. The arithmetic operators take precedence over the relational operators. So one can replace the objects by arithmetic expressions without any ambiguity. The result of the relational operators is logical (true or false).

5.3 Logical Operators

Logical operators operate on logical objects. You will need the logical operators when you will be using the results of relational operators in computing. For example, if you want to compute certain thing when (say) $a > b$ and $c > d$, you would use relational

operators for computing whether $a > b$ and $c > d$ and use the results of the two in logical and operation. The logical operators are logical and (.and.), logical or (.or.), logical equivalence (.eqv.) and logical nonequivalence (.neqv.).

5.4 Precedence of Operators

Among different kinds of operators, arithmetic operators have highest precedence and logical operators have lowest precedence.

6 Examples and Problems (Set I)

Armed with the methods of definition of objects and the knowledge of operators, we are now in position to write simple programs. At this stage, programs will consist of single units as we have not learnt about subroutines and functions. We have also not learnt about some useful constructs which allow us to define repetitive tasks. You still need to know how one can communicate with the computer during the execution of the program. That is, you need to supply the data to the computer so that it can continue computation and the computer should be able to show you the results of the computation. This communication requires the knowledge of I/O (or Input/Output) statements. The simplest I/O statements are,

```
read *,x,y,z
```

for reading from the standard device (the key board) and

```
print *,x,y,z
```

for printing on the standard device (the computer screen). Here x etc are the

variables whose values are to be read or printed. The read and write statements can be made more sophisticated to read from/write to files, write the information in a nice format etc. We shall discuss that later.

Now we shall do some exercises on writing simple fortran programs.

1. Find largest integer that can be stored in fortran. Begining with an integer, keep on multiplying it by 7 and print the result. You have to do this several times. At some stage you will start getting funny results. Your program should read an integer and print it. Then starting with that integer, it should keep on multiplying the result by 7 and printing it. (You could choose any number instead of 7 for multiplication. 7 is not too small so that we have to write a program without do construct and not too large as well.)

2. Finding smallest and largest real (and double precision) number that can be represented in a computer. For smallest number, you can start with a small number (say 10^{-3}) and keep on dividing it by 3 several times. Eventually you will have funny results. For largest number, keep on multiplying a large number (say 10^{10} and keep on multiplying it by (say) 1838. (any four digit number will do).

3. Given values of a, b, c compute the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

. The values of a, b and c may be real but the root may be (and therefore should be declared) complex.

4. This example highlights round-off errors in computer calculations. As you know,

the derivative of a function at a point is defined as

$$f'(x) = \mathrm{Lim}_{\delta x \to 0} \frac{f(x+\delta x) - f(x)}{\delta x}$$

. Compute the derivative of e^x at x = 2 using this method. We know that the derivative is e^2. Compute the derivative using δx going from 10^{-1} to 10^{-7} for single precision calculation and 10^{-1} to 10^{-14} for double precision calculation. Compare the result with the exact value.

5. This is another example of round off errors. Here we shall compute e^x using series

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} \cdots$$

. We shall compute the result for $x = \pm 2$ and keep terms unto seventh power and compare the result with exact value. Compare the accuracy for positive and negative values of x. Note that for positive x, all the terms add up where as for negative x, the alternate terms change sign. As a result, there is large cancellations and larger round off error

7 Do, If And Case Constructs

One can, in principle, use the material discussed in the previous subsections and write fortran programs to do any calculation you want. However, such programs will be very long for any reasonably complex calculation. Consider a calculation where you are computing sin(x) using a series expansion, keeping some 10 terms. As you know, the nth term is of the form $(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}$. One can indeed write a fortran program using what we have learnt but it will be a rather long program. In order to reduce

such repeative tasks to a simpler code, two types of constructs are devised in fortran. One we shall discuss in this subsection and the second in the following subsection.

Consider a calculation where you have to compute something for a number of values of a parameter. For example, you have to compute the Legendre polynomials for different values of it's argument. We can certainly write a program to compute it for the argument equal to 0, 1, 2, etc. But it would be a very long program. In such cases one can use do-end do construct. Similarly, consider the case when you have to do a calculation when a certain quantity is positive and another calculation when it is negative. For this if-then-else construct is very handy. Note that these two constructs, by themselves, form a complete sub-unit of a program. Thus a transfer of execution into the do or if constructs is not allowed. One is, however permitted an exit from these constructs at any place.

7.1 Do Construct

As a concrete example of do construct, consider a calculation of factorial of an integer. The definition of a factorial is $n! = n \times (n-1) \cdots 2 \times 1$ and this can be written as a recurrence relation $n! = n \times (n-1)!$. Using this relation, one can compute factorials for small enough values of n. One can use the do construct which permits one to compute the factorial using a compact fortran code.

The essential structure of the do construct is

```
do i=n1,n2,n3
   fortran statements
end do
```

Here i is the do-loop index (could be real number but usually one chooses it as an integer), n1, n2 and n3 are initial value, final value and increment in i and the

fortran statements (at least some of them) depend on i. Otherwise, why have the do construct anyway. n3 can be negative and then n1 must be larger than n2. The do-loop is executed at least once. For the calculation of factorial, the structure of the code is

```
fact=1.
do i=2,n
 fact = fact*i
end do
```

Note that if the increment (n3) is not mentioned, it is taken to be unity. The factorial has been defined as a real number (not integer) because of integer overflows. In the above code, n can be any integer larger than 1. For n = 2, the do-loop is executed once.

One can embed do constructs within do constructs and so on. It is therefore a good practice to indent the fortran statements within a do construct so that a block of these statements is clearly marked for easy reading as well as for finding errors in the program.

7.2 If construct

As mentioned earlier, the if construct is used when there is a fork in the computation. It should be obvious that while using if construct, one needs relational operators as well as logical operators. The structure of the if construct is, if certain condition is satisfied, a certain fortran statements are executed. Otherwise certain other statements

are executed. An example is the calculation of the roots of a quadratic equation

$$ax^2 + bx + c = 0. \tag{1}$$

As we know, the roots are

$$x = -\frac{b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2}$$

For real coefficients (a, b, c), the roots are real if $b^2 \geq 4ac$. Otherwise these are complex. That is because the square root of a negative number is imaginary. Now, if we don't want to do complex arithmetic but want to keep track of when the roots are real and complex, one can use the if construct. The fortran code in this case would be

```
if( b**2 . ge. 4.*a*c)then          root1=-( b + sqrt( b**2 - 4.*a*c))/2./a
 root2=-( b - sqrt( b**2 - 4.*a*c))/2./a
else
 print *,'complex roots'
 rootr = -b/2/a.
 rooti = sqrt(4.*a*c - b**2)/2./a
end if
```

The meaning of the code is obvious. The if statement (the first line above) can be more complex. It may be something like if(a.lt.b.and.c.gt.d) and so on. The basic thing is, we evaluate a certain condition. If it is true, certain statements are executed. Else, some other statements are executed. Further, the statement else may be replaced by else if (another condition) so the if construct may look like if-then-else if-else-end if (with possibly more else if's). As in case of do construct, if construct also forms a block and therefore it is a good practice to indent the fortran statements

within if construct as shown. Also, entry into the if construct is allowed only at the if statement.

7.3 Case construct

The case construct is allowed in fortran 90 but not in fortran 77. Essentially, it has been borrowed from C. It is similar to if construct but has larger scope. Consider that different calculations are to be done depending on the value of certain expression *expr*. The structure of the case construct is

```
select expr
case ( select1)
  expr1
case ( select2 )
  expr2
end select
```

The quantities *select1*, *select2* etc must be of same type as the quantity *expr*. The statements between two case statements (or the case statement and end select statement for the last select) are executed if the value of *expr* equals the value of the quantity in the bracket following the first select. Thus select construct is similar to the arithmetic if which was available in older versions of fortran. Again, the case construct forms a block and an entry into this block is permitted only at the select statement. It is useful to indent the fortran expressions occurring in the select construct as shown.

7.4 Go To And Continue Statements

In fortran, the statements are executed sequentially. So the program execution begins from the first line and continues till the stop (or return) statement is encountered. Sometimes it may be necessary to skip certain statements and transfer the control to a statement some statement doen the line or go back to some previous statement. For example, if you are reading from a file and a priori you don't know the number of lines in the file. In that case, you would be reading one line at a time and executing some statements. Here the go to statement is useful. It passes the control back to the read statement after execution of the statements and when one encounters the end of file, the control is transfered to the statement following the statements being executed for each read. This is like a do construct but here there is no loop index. To begin with, we can not use do construct because we do not know the number of lines to be read. In go to statement, the statement where the control is transfered is labeled by a number. So, the go to is used as follows:

expr1
if(a.lt.c)go to 10
expr2
10 continue
expr3

In the code above, if a is less than c, the control is transfered to a statement labeled 10. This process skips the fortran code in *expr2*. If a is greater than or equal to c, the code *expr2* is executed. The statements in *expr3* are executed for all values of a and c.

As mentioned above, the go to statement may transfer the control to statements preceding the go to statement also. This aspect of go to statement (jumping from

one place to another in the fortran code) is not very nice from the point of view of readability of the code as well as for finding errors in the program. One may have statements where the control is transfered to part of the program which is hundreds of lines away (before or after the statement). If there are many such statements, following the logic of the program becomes very difficult. Therefore one should use go to statements only in extreme situations where it is practically impossible to find an alternative.

The continue statement is do nothing statement. Usually it is used in conjunction with go to statement. Usually, the go to statement transfers control to continue statement. It is used more as convenience and usually it is possible to eliminate it.

8 Examples And Problems (Set II)

In this set, we shall consider the application of go to and if constructs and go to statement.

1. Given a largest number N, compute the primes between 1 and N. By definition, an integer which is not divisible by any integer except 1 and itself is a prime. To check whether an integer is a prime or not, we need to divide by a prime and see if any remainder is left. A little thinking will show that we need to test the division by all primes smaller than the square root of N. You can write an inefficient but simple program which tests division by **all** integers smaller than $\sqrt{N}$. It is best to use integer division to test whether the remainder is zero or not (If (N/m)*m is equal to N the remainder is zero). Or you can write an efficient program which prepares a table of primes while it is testing the division. The bonus in the second case is you have a table of primes after

testing. For large N, the second program will be faster as well.

2. The coefficients of binomial expansion are defined as $C(n,m) = \frac{n!}{m!(n-m)!}$. One can use this definition. There is an alternate definition due to Pascal (called Pascal triangle). The successive rows of the Pascal triangle are for successive orders of the binomial expansion and the elements in a row are given as a sum of two elements in the preceding row closest to the element to be computed. First few rows are shown in the accompanying table.

						1						
					1		1					
				1		2		1				
			1		3		3		1			
		1		4		6		4		1		
	1		5		10		10		5		1	
1		6		15		20		15		6		1

Use the Pascal's algorithm to compute the Pascal traingle for given N and print it. (Actually, this algorithm works better than using factorials. That is because you need to compute three factorials and multiply them to get one coefficient. In this algorithm you add two numbers.)

3. Write a program which reads three coefficients of a quadratic equation and prints the roots. Don't use complex arithmetic but use if statement. You should distingwish the cases when two roots are equal, two roots are real and two roots are complex.

4. Write a program to compute Legendre polynomeal of order N for any argument

between -1 and 1. The program should read N and x. If x is outside the scope, it should print so and exit. Otherwise it should compute the Legendre polynomeal $P_N(x)$. We have $P_0(x) = 1$, $P_1(x) = x$, $P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$ and a recurrence relation $P_N(x) =$. So the program should use the recurrence relation for $N \geq 2$.

5. Any arbitrary rotation can be represented by three Euler angles. These are defined as three successive rotations along z axis through angle α, followed by a rotation along y axis by an angle β and a rotation along z axis by an angle γ. A rotation matrix (a three-by-three orthogonal matrix) for each of these rotation can be defined (look into a classical mechanics book for it). Write a program which reads in the Cartesian coordinates of a vector and the three Euler angles. It then operates the three Euler rotations on the vector read in and prints the components of the vector after rotations.

9 Subroutines, Functions And All That

What we have learnt is sufficient to write fortran programs to do most of the calculations. However, there would be a lot more repetitions if we use what we have learnt so far. For example, consider a situation in which you are wanting to have complex numbers with double precision. Then you have to write code for addition, subtraction, multiplication and division of two complex numbers, since the machine-defined complex numbers are single precision. In fact you will need to define the complex numbers as a pair of double precision numbers (say zx1 and zy1) and define operations on them. So the equations for addition of two complex numbers will

be

$$\begin{aligned} zx3 &= zx1 + zx2 \\ zy3 &= zy1 + zy2. \end{aligned} \tag{3}$$

Then every time you are doing addition of two complex numbers, you will be writing two such equations. This will work but the computer code will become bulky. If you can define such equations as independent sub-units of a program and use these sub-units whenever needed, the code will be smaller as well as more transparant. This is what subroutines and functions do.

One more advantage of dividing the computer code into subroutines and functions is these units can be written and tested independently and then used in many places. Because these subunits are small in length and are performing small tasks, it is much easier to find errors and fix them. And once you have done this, these can be used as black boxes, top be used whenever needed. That increases efficiency of programming. This is, in fact, an example of modular programming, which is the 'in thing' these days.

This brings us to general methodology of computer programming. The rule is, don't rush to the terminal when you are to solve a problem on computer. Think about the problem and divide the complete task into a number of sub-tasks which can be done independently. Plan how the final program will look like when developed (number of sub-tasks, how they are linked to each other etc). Then find out whether you have developed codes for some of the tasks already or whether you can get such codes from somewhere (friends, libraries). Then, develop the subtasks not developed already, test them and finally integrate the whole thing into a single program. However if you are taking codes from other sources, make sure to understand the limitations

of the code and make sure that the parameter range of your interest is within the limitations of the borrowed code. Libraries usually mention such limitations. Always be wary of the hand-me-down codes which usually not well documented.

There are basically two types of modules used in fortran. One is the subroutine and the other is the function. These are the two modules available in fortran 77. In fortran 90, one can have a more complex structure. One can combine several functions and subroutines to form a module. One may include functions and subroutines as a part of the program (main program, function or subroutine). These are called as the internal functions or subroutines. Others are called external functions and subroutines. In fortran 77, in-line function definitions are allowed. As an example, consider the calculation of relativistic energy which is square root of the sum of squares of the momentum and the mass of the particle. Coding it every time will require some time. But we can define an in-line function (say ener(p,m)=sqrt(p**2+m**2)) just before the first executable statement. In the body of the program, we can then use ener(a,b) to compute the energy of a particle of momentum a and mass b. In this case the compiler inserts the definition of the function where one types ener(a,b). This is better than defining an external function as it is faster.

9.1 Main Program

Each executable program should have at least one and only one main program. (It's name, ofcourse, need not be main). The structure of the main program is discussed in the begining of this section. In complex codes, the main program usually defines global quantities, does much of input-output work and much of the computations are done in subroutines and functions.

9.2 Functions

Functions are like mathematical functions. The result of the function calculation is a value which is stored in an object having the same name as the function. Thus, the function returns one value. It may depend on a number of arguments. As a concrete example, consider the calculation of the relativistic energy. If one defines a function for this, it would have the following form.

```
function ener(a,b)
implicit none
real*8 a,b,ener
ener=sqrt(a**2+b**2)
return
end
```

The general structure of the function is: the function declaration, declaration of variables used as well as passed as paremeters (including the function name), body of the function (where the calculation is performed), a return statement signifying the end of the calculation and return of the control to the place where the function is called, and finally, the end statement signifying the end of the function module. As an option, one may include the name of the function after the end statement. The return statement may appear anywhere in the code. It may even appear more than once. The calling sequence in the module where the function is called is as follows:

```
e1=ener(p1,m1)
etot=etot+ener(p1,m1)
```

The program segment above shows two different ways of calling the function. In the first line, the result of the function call is stored in the variable e1. In the second line, the result is directly used in another calculation.

9.3 Subroutines

The subroutine is somewhat more complex than the function. The subroutine may return more than one object after the execution. These objects are passed as output arguments. Thus, the subroutine may have input arguments as well as output arguments. Obviously, one can convert a function into a subroutine by including the result as a single output argument. The structure of the subroutine is similar to the structure of the function, except that one does not declare the name of the subroutine as a variable. Another difference between the function and subroutine is the arguments of the function are not expected to be changed in the function where as the arguments of the subroutine may change. As an example, here is a subroutine which rotates a vector (specified as an array of three numbers representing three coordinates) about z-axis through angle th. The rotation matrix (M) for this rotation is

$$M = \begin{pmatrix} cos(th) & sin(th) & 0 \\ -sin(th) & cos(th) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4}$$

```
subroutine zrotate(th,a)
implicit none
real*8 th, a(3), matrix(3,3), cs, sn, b(3)
integer i,j
cs=cos(th)
sn=sin(th)
do i=1,3
 do j=1,3
```

```
    matrix(i,j)=0.
   end do
   b(i)=0.
  end do
  matrix(1,1)=cs
  matrix(2,2)=cs
  matrix(1,2)=sn
  matrix(2,1)=-sn
  matrix(3,3)=1.
  do i=1,3
   do j=1,3
    b(i)=b(i)+matrix(i,j)*a(j)
   end do
  end do
  do i=1,3
   a(i)=b(i)
  end do
  return
  end
```

The subroutine is called as given below:

hspace*1.0cmcall zrotate(angle,vector)

Here angle is the angle through which a vector is rotated and vector is the vector which is rotated.

Note that the names of the variables in the calling program and subroutines and

functions can be different (most of the time they are!) but their types must be same. For example, in case of the subroutine defined above, the angle (angle in calling program and th in subroutine) is a double precision scalar and the vector (vector and a respectively) is an array with three double precision elements. If the type or dimention of the objects differ one may either end up with wrong results or (more likely) some error with core dump.

9.4 Internal Subprograms

The concept of internal subprograms is not allowed in fortran 77. Internal subprograms are same as functions and subroutines except that they are part of a bigger subprogram (main program or function or subroutine). That is they come before the end statement of the parent module. The statement before the internal subprogram declaration must be 'contains' statement listing the subroutines and functions contained in the module. Also, the end statement of the internal subprograms must give the name of the subprogram.

1. Write a function to compute the Legendre polynomeal of a given order and argument. The program should use the recurrence relation defined earlier. It should use if statement to compute the Legendre polynomeal for order 0, 1, 2 and rest (doesn't make sense to use recurrence relation for order 0, 1 and 2). The function should return only one value.

2. Write a function to compute the spherical Bessel function of order upto 20 and argument upto 20. For small arguments, the series expansion converges rapidly. Question is what cutoff in argument should be used. The rule of thumb is the argument should be less than or of the order of the order (except for 0 and 1,

the functions being sin(x)/x, sin(x)/x**2 - cos(x)/x. So for these, you simply compute these functions.). You must keep sufficient number of terms to ensure sufficient accuracy. For argument larger than the order, recurrence relation is best. You must test the function you have for possible errors.

3. Write a subroutine to compute the rotation of a vector, given the three Euler angles. We have a subroutine for a single rotation along z-axis. You have to generaliae that subroutine to do three successive rotations.

4. Write a function to compute factorial of a number. Using the standard definition of the factorial is fine for small values of the argument (typically less than 20). For larger values the direct method is inefficient (imagine multiplying 10^6 numbers to get factorial). One should use the Sterling-like approximation which is acurate enough. The formula is

5. Write functions to compute addition, subtraction, multiplication and division of two complex numbers. Choose the real and imaginary parts of the numbers to be double precision.